National Library of Canada
Acquisitions and Bibliographic Services Branch
395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale du Canada
Direction des acquisitions et des services bibliographiques
395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file *Votre référence*

Our file *Notre référence*

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Canada

University of Alberta

SLOTT — A Source Level Oriented Timing Tool

by

Ladislav Hala

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of Master of Science.

Department of Computing Science

Edmonton, Alberta
Fall 1992

National Library of Canada
Bibliothèque nationale du Canada

Canadian Theses Service
Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-77110-0

Canada

UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR:	Ladislav Hala
TITLE OF THESIS:	SLOTT - A Source Level Oriented Timing Tool
DEGREE:	Master of Science
YEAR THIS THESIS GRANTED:	1992

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as hereinbefore provided neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.

Ladislav Hala

Permanent Address:
Box 60191, U of A Postal Outlet
Edmonton, Alberta
T6G 2S5
Canada

Date: October 7, 1992

Sloth: Another of those confounded hypotheticals! Why are the rest of you always running off into your worlds of fantasy? If I were you, I would stay firmly grounded in reality. "No subjunctive nonsense" is my motto.

Douglas R. Hofstadter
(*Gödel, Escher, Bach: An Eternal Golden Braid*; Chapter XVIII, Contrafactus)

UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled

SLOTT - A Source Level Oriented Timing Tool

submitted by Ladislav Hala

in partial fulfillment of the requirements for the degree of Master of Science.

Dr. H. Zhang (Supervisor)

Dr. P. Rudnicki (Committee Member)

Dr. J. Mowchenko (Committee Member)

Date: August 17, 1992

Abstract

Real-time systems, especially the hard real-time systems, execute under stringent time constraints as they are scheduled to finish tasks on time, *i.e.* , before the next interrupt comes; otherwise, degrading the system performance to the point of a partial or the complete system failure. To satisfy timing requirements, it is necessary to test and evaluate such systems under all conditions. Conventional tools and testing methods have been proven to be inaccurate, inadequate and time consuming.

Recently, there has been a development in the area of source level timing that tries to predict the execution time directly from the source code. This thesis outlines one such scheme and presents an implementation of a timing tool — the *Source Level-Oriented Timing Tool* (SLOTT). Our approach extends the scope of an already existing development system by adding the timing functionality to the compiler. The main advantage of using the timing tool is in obtaining the timing values immediately after the compilation is finished.

The tool consists of two parts, one that gathers the timing information, the other that analyzes it. The former is implemented as a part of the GNU C compiler. It collects the timing information of each instruction that is generated by the compiler, records it in the appropriate structure and outputs the timing values into an intermediate file. The second part, the timing analyzer, then reads the intermediate file and produces the timing output. The analyzer is implemented on its own and can be customized.

The tool can time any part of the source code. The code size can range from a single line to the whole function. The tool recognizes several markers that identify boundaries for the timed code, bound loops and specify program flow. These markers are ignored by the compilers that do not implement SLOTT.

The thesis describes the timing methodology of SLOTT, its design, implementation, testing and evaluation.

Acknowledgments

I would like to thank all those that helped me in making this thesis better, namely my brother Peter, my supervisor Professor Hong Zhang, committee members Dr. Piotr Rudnicky and Dr. Jack Mowchenko, and finally a friend of mine – Michael Carbonaro. All of them provided valuable comments for improvement, either of technical nature, or just by proofreading my drafts and final versions, or both.

Also, I would like to thank NSERC for providing me with a scholarship that supported me financially during my studies.

Finally, and most importantly, many special thanks to my parents and brother that supported, guided and encouraged me in many ways during all of my studies. This work is dedicated to them.

Table of Contents

Abstract iv

Acknowledgments v

List of Figures x

List of Tables xi

List of Graphs xii

1 Introduction 1

1.1 Background and Motivation 1

1.2 Basic Issues of Real-time Computing 2

1.2.1 Real-time Systems 2

1.2.2 Execution Time and Response Time 2

1.3 Thesis Objective 3

1.4 Thesis Outline 3

2 Reliability Guarantee of Real-time Systems 5

2.1 Languages Used for Real-time Programming 5

2.1.1 Predecessors of Real-time Languages 6

2.1.2 Early Real Time Languages 6

2.1.3 Process Control Languages 7

2.1.3.1 PEARL 7

2.1.3.2 Forth 7

2.1.4 Concurrent Languages 8

2.1.4.1 Modula and Modula-2 8

2.1.4.2 Ada 8

2.1.5 Non Real-time Languages 8

2.1.6 Survey Summary 9

2.2 Time Evaluation and Design Tools 9
2.2.1 Profilers 10
2.2.2 Simulation 11
2.2.3 Source-level Time Analyzing Tools 11
2.2.3.1 Details of RT Euclid, MARS-C and SLTS 12
2.2.3.2 Comparison of RT Euclid, MARS-C and SLTS 13

3 Timing Tool Basics 14

3.1 Methodology of the Timing Tool 14
3.1.1 General Aspects 14
3.1.2 Program Markers 15
3.1.3 Timing Blocks 16
3.1.4 Bounding of Looping Constructs and Recursive Function Calls 16
3.1.4.1 The Need for Loop Bounding 16
3.1.4.2 SLOTT Bounding Markers 17
3.1.4.3 Bounding Execution Time 18
3.1.4.4 Recursive Function Calls 19
3.1.4.5 SLOTT's Automatic FOR Loop Bounding 19
3.1.5 Program Flow Statements 20
3.1.6 Informing SLOTT of Program Flow 21
3.1.7 Inline Assembly 22
3.2 Theoretical Approach to the Execution Time Prediction 22
3.2.1 Basic Constructs of a Procedural Language 22
3.2.1.1 Simple and Compound Statements, and Functions 22
3.2.1.2 Conditional Statements 23
3.2.1.3 Loops 24
3.2.1.4 Execution Time Constructs Summary 25
3.3 Timing Process 26
3.3.1 Structure of Timing Blocks and Nodes 26
3.3.2 Maintenance of Timing Structures during Timing Process 28
3.3.3 Steps of the Timing Process 29
3.3.4 Treatment of Control Statements 34
3.4 Execution Time Analyzer 34
3.4.1 Reasons for Compiler Independent Implementation 34
3.4.2 Analyzer's Output 35

4 Implementation of SLOTT **38**

4.1 Compiler Modifications....38

4.1.1 Compiler Fundamentals....38

4.1.1.1 Compilation Process....39

4.1.1.2 Parsing Process....40

4.1.2 Outline of Compiler Modifications....41

4.1.3 Lexical Analyzer Modifications....42

4.1.3.1 #pragma Approach....42

4.1.3.2 Reserved Word Approach....42

4.1.4 Parser Modifications....42

4.1.5 Code Generator Modifications....43

4.2 Machine Analyzer....44

4.2.1 Instruction Execution Timing Calculations....44

4.2.2 Recording Process....47

4.3 GCC Modification Aspects....48

4.3.1 Overview of GCC....49

4.3.2 SLOTT's Program Structures and its Constructs in RTL....50

4.3.3 Timing Process During GCC Compilation....50

4.4 Timing Analyzer....51

5 Evaluation of SLOTT **53**

5.1 Choosing the Correct Hardware System....53

5.2 Evaluation of MC68000 Hardware....54

5.2.1 Test Programs....55

5.2.2 Test Results....55

5.3 Factors That Affect Execution Time....56

5.3.1 Instruction Alignment....57

5.3.2 Instruction Overlap....58

5.3.3 Effect of Operand Alignment....58

5.3.4 Effect of Cache....59

5.3.5 Summary of the Results....60

5.4 Testing General Purpose Procedures....61

5.4.1 Tests Involving only the CPU Instructions....61

5.4.1.1 Significantly Large Programs....61

5.4.1.2 Small Programs 61
5.4.1.3 Cache OFF Test Results 62
5.4.1.4 Cache ON Test Results 63
5.4.2 Tests Involving CPU and FPU Instructions in Same Proportions 64
5.4.2.1 Test Programs 64
5.4.2.2 Test Results 64
5.4.3 Tests Involving Mostly FPU Instructions 65
5.4.3.1 Test Programs 65
5.4.3.2 Cache OFF Test Results 65
5.4.3.3 Cache ON Test Results 66
5.4.4. Summary of the Test Results 66
5.5 Accuracy of SLOTT 69
5.5.1 MC68000 System 69
5.5.2 MC68020 System 70
5.5.2.1 COFF Case with Worst Data Book Values 70
5.5.2.2 CON Case with Cache Data Book Values 71
5.5.2.3 COFF Case with Normalized Values 71
5.5.3 Summary 72
5.6 Comparison of SLOTT with Other Systems 73

6 Conclusion **76**

References **80**

Appendix **85**

List of Figures

3.1 Incorporation of SLOTT into a compiler 15

3.2 Timing structures used by SLOTT 27

3.3 Creation and linking of timing blocks and nodes 31-33

4.1 Basic blocks of a compiler 39

4.2 Parser generation from the grammar rules 40

4.3 Parser in action producing a parse tree of tokens 41

4.4 Parts involved in the modification of a compiler 41

4.5 Determining the execution time of an instruction from table values 48

List of Tables

3.1 Constructs of a procedural language and their timing formulas 25

5.1 Execution time of MC68000 test programs. 55

5.2 Testing the use of the average and maximum timing values for the MC68000 multiplication instruction. 56

5.3a Execution time of CPU-only instructions (cache OFF case) 62

5.3b Execution time of CPU-only instructions using normalized execution values. (cache OFF case). 63

5.3c Execution time of CPU-only instructions (cache ON case). 63

5.4 The execution time of programs having equal number of CPU and FPU instruction cycles. Both the data book and normalized values are given for the COFF case; the CON case uses only the data book values 64

5.5a Execution time of FPU-oriented programs (cache OFF case). Both normalized and data book prediction values are summarized. 65

5.5b Execution time of FPU-oriented programs. (cache ON case) 66

5.6 Range of errors, offset error and average prediction error of the MC68000 system. 70

5.7a Range of errors, offset errors and average prediction errors of the *worst* data book values under COFF setting. 70

5.7b Range of errors, offset errors and average prediction errors of the *cache* data book values under CON setting 71

5.7c Range of errors, offset errors and average prediction errors of the *normalized* data book values under COFF setting 71

5.8 Comparison of SLOTT with other source level timing tools. 75

List of Graphs

5.1 The effect of loop size on the CON execution time. 60

5.2a Range of prediction errors using data book timing values (COFF errors). 68

5.2a Range of prediction errors using data book timing values (CON errors). 68

5.2c Range of prediction errors using normalized values. 69

5.3a Percentage errors for all MC68000 test cases. 72

5.3b Percentage errors for MC68020 test cases. 73

Chapter 1

Introduction

This chapter gives a brief background and motivation of the thesis, presents basic terms and definitions of real-time computing, describes why timing is of crucial importance for the design of real-time systems, and outlines the goals of the thesis.

1.1 Background and Motivation

Since the appearance of the first microprocessor in the early seventies there has been a continuous trend toward designing capable and "clever" electromechanical computer controlled devices and systems. However, as the technology is getting more powerful, computer controlled systems are also becoming more complex. Issues, never considered before, arise as new intricate hardware and software concepts are introduced. It thus becomes increasingly more important to keep the technology under control.

A computer controlled car runs people over when it unexpectedly accelerates, a fly-by-wire state of the art aircraft refuses to release control to a human pilot and the aircraft crashes and a software error causes a stationary robot to move suddenly out of its pre-programmed bounds and a nearby worker is crushed to death. These are just a few examples of how technology can inadvertently change our admiration to cynicism and distrust [Burns90].

The questions of hardware and software efficiency, versatility, flexibility and reliability are becoming increasingly important. The true control of technical processes, also referred to as real-time systems [Halang91], puts a lot of responsibility on the programmer's shoulders. Consequently the last two decades have seen the birth of a new computing sciences discipline called *software metrics*. It deals with measuring

software quality, performance testing and optimization [Gilb77]. The research focus of the real-time design during the last decade has been the performance guarantee and reliability of systems, *i.e.*, the guarantee that the real-time tasks finish within defined limits that are imposed by the system.

1.2 Basic Issues of Real-time Computing

1.2.1 Real-time Systems

A *real-time system* is one which has to respond to externally generated input stimuli within a finite and specified period [Young82]. A fundamental property of a real-time system is that some or all of its inputs come from the outside world asynchronously with respect to any work that the program is already performing. The program must be able to interrupt its current activity, respond and resume the previous activity. In general sense, this definition covers a very wide range of computer applications and it is often desirable to distinguish the real-time systems by another property - the task scheduling deadline.

Hard real-time systems (HRT) are those where it is absolutely necessary that responses occur within the specified deadline [Burns90]. The time at which each task must finish a round of processing must be estimated and the design must pay close attention to the execution timing of each process and task. For an HRT system a late response due to the insufficient execution time is as unacceptable as an incorrect response. Improper design of an HRT system may result in the degraded or unacceptable performance sometimes in a partial or a complete system failure. However, not all real-time systems demand such strict constraints.

A system in which the response time is important but task deadlines can be occasionally missed without affecting its functionality is referred to as the *soft real-time* (SRT) system [Ripps89].

A data acquisition system and a UNIX operating system are typical examples of soft real-time systems; a flight control system, a nuclear reactor process control system and a production robot manipulator are typical examples of hard real-time systems.

1.2.2 Execution Time and Response Time

Execution time and *response time* and their measurement is of crucial importance when designing either SRT or HRT systems and tasks. Unfortunately, given the nature of the problem, it is difficult to design systems which will guarantee that all timing requirements will be met under all possible loading and under all internal and external conditions.

To meet the time constraints real-time systems are often constructed using processors and hardware with considerable spare computing capacity, so that the worst-case can be handled. However, to achieve the system reliability various software or hardware tools must be used to test and perfect the system.

1.3 Thesis Objective

The thesis will outline tools and schemes that have been used to gather timing information about real-time programs. Moreover, the thesis will focus on one such tool and its methodology that help to determine the execution time of a program — the *Source Level Oriented Timing Tool* (*SLOTT*), operating appropriately at the source level. A number of different test procedures, that evaluate SLOTT, will be implemented. The results of these testing procedures will be analyzed to measure SLOTT's accuracy. To demonstrate the tool's flexibility, adaptability and modularity, the thesis will describe and test its implementation on the 680X0 class hardware.

1.4 Thesis Outline

Chapter 2 starts with an overview of languages used for real-time programming and outlines conventional methods and tools for software development, debugging, and troubleshooting. The chapter concludes with the review of the latest developments in the performance and reliability guarantee analysis research.

Chapter 3 presents the theoretical aspects of collecting the timing data, and all language structures that have to be considered. SLOTT's features and structures are discussed along with the process that describes how SLOTT gathers the timing information.

Chapter 4 goes into implementation details with regards to any compiler. It gives an introduction to compilers by describing how their individual components work, and what modifications have to be performed to incorporate SLOTT inside the compiler. Additionally, compiler independent tool parts will be described. Also, implementation aspects, that relate to a widely available C compiler – the GNU C compiler, will be given.

Chapter 5 presents the evaluation results of SLOTT that have been gathered by running several test programs. These programs are aimed at certain groups of CPU instructions and specific CPU properties; consequently, they are very specific to what they are trying to accomplish. As real tests, several of real-time

and of general nature procedures have been tested and then compared with the actual execution time values. Furthermore, a number of the procedures were tested on a different hardware system to demonstrate the flexibility of SLOTT.

Chapter 6 concludes this thesis by outlining several issues that have not been covered in this work. It shows further steps that might be undertaken in the future to enhance the timing utility.

Chapter 2

Reliability Guarantee of Real-time Systems

The purpose of this chapter is to introduce some real-time systems that have been used by the scientific community and industry, identify their common deficiency — the lack of timing aspects and *schedulability analysis* , and present tools that have been used to compensate for this deficiency of time evaluation. The schedulability analysis is a term introduced by Stoyenko [Stoyenko87], [Klingerman89], [Halang91]. A program is said to be schedulability analyzable if the program deadlines can be determined at compile-time. The tools can be classified into two categories — (1) those that arrive at the timing values by testing the program and (2) those that predict the execution time directly from the source code. The *Source Level Oriented Timing Tool* (SLOTT) is an example of the latter group.

2.1 Languages Used for Real-time Programming

Real-time languages are differentiated from other programming languages in several aspects. They must be reliable (or secure), and should allow multiprogramming and synchronization of several processes. They should also present themselves as maintainable programming environments, able to handle large projects. Since the real-time programs manipulate and control hardware, they should have easy access to the hardware they handle. The real-time programs are designed to run for the lifetime of the systems they control. Consequently, it is not surprising that the most important aspect of any real-time system is the knowledge whether the tasks can execute within the specified time constraints. To obtain this information, the programs must be fully tested or they must be designed on systems that allow time analysis.

Real-time languages can be divided into several categories: the early real-time languages, process control languages and concurrent languages. Even though some languages are not classified as real-time, they lend themselves as good environments for designing some real-time applications.

2.1.1 Predecessors of Real-time Languages

Initially, the real-time systems were programmed in low level assembly languages because they were the only practical way of achieving efficiency. Early real-time applications often ran with "ad hoc" programs developed for one specific purpose [Ripps89]. Cooling[1991] points out that *Sequential* languages (FORTRAN, COBOL, ALGOL) were the first widely used high level languages but they have generally been ignored for real-time work.

The mass production of microprocessors prompted the development of high-level languages aimed at the direct microprocessor programming and easy "ROM-ability" [Leigh88]. Microprocessor manufacturers developed their own languages designed specifically for their line of microprocessors such as PL/M and MPL [Cooling91].

The special problems of real-time programming were not fully appreciated until the 1960's when the first real-time languages appeared. Since then, many real-time languages have been developed in a search for the "ultimate" language.

2.1.2 Early Real Time Languages

The early real-time languages were concerned with the real-time aspects like concurrency, security, hardware access while they tried to retain some of the features of other current languages like modularity and maintainability. They also strived for efficiency. However, none have been concerned with the timing aspects.

The earliest real-time language was JOVIAL created in 1959 as a standard for the U.S. Air Force. It is largely based on ALGOL. Unlike ALGOL, it supports assembly coding, arrays and records. However, it lacks: (1) exception handlers, (2) multitasking, (3) modularity and (4) provisions for schedulability analysis [Halang91].

Another language that borrows from ALGOL is CORAL 66 which became the standard defense language in the UK [Cooling91]. Similarly to JOVIAL, it does not support modularity and schedulability.

RTL/2 (Real Time Language Two), developed in the early 1970's, was considered by many a major improvement over CORAL 66. Based on ALGOL-68, it was designed for industrial process control. Even though it supports some concurrency and modularity, it is weakly typed, and has no provision for

schedulability [Halang91]. RTL/2 was used mainly within the chemical industry, but otherwise it has not made a significant impact on other real-time applications [Cooling91].

Currently, there is little future for the above languages since they are being replaced by more advanced languages like Ada, Modula, PEARL and even C.

2.1.3 Process Control Languages

Process control systems are sometimes referred to as "command and control" systems. The primary objective of such systems is to monitor the sensory input from the outside world, evaluate the information and react accordingly by controlling the external components [Yourdon72]. The main characteristic of such systems is their fast response time. Even though there have been many languages designed for process control, this section will outline only two, one that shows the most potential, the other that stands on its own.

2.1.3.1 PEARL

PEARL (*P*rocess and *E*xperiment *A*utomation *R*eal-time *L*anguage) was designed in the early 1970's by a group of German researchers and is now widely used throughout German industry. It was designed by a team of electrical, chemical and process control engineers based on their practical experience in industrial automation [Halang91].

PEARL is a Pascal-like language which provides low level peripheral constructs, concurrent features for task scheduling and the expression of time constrained behavior. Unfortunately, PEARL has a number of shortcomings: (1) the lack of well-structured synchronization primitives, (2) unstructured exception handling facilities, (3) no means to interrogate tasks and resource states (4) and insufficient provisions for schedulability analysis. Halang[1991] indicates that recent research is underway to enhance PEARL to PEARL 90 thereby eliminating many of the above deficiencies.

2.1.3.2 Forth

Developed in the late 1960's and commercially available since the early 1970's, Forth has been described as "a language apart" [Dettmer88]. Forth is a language, an operating system, a set of tools, and a philosophy [Brodie82]. It was created because as the author, Charles H. Moore, put it - "the traditional languages were not providing the power, ease, or flexibility" he wanted [Brodie81]. Forth became quite popular and is used in a number of process control applications. Forth differs from other real-time languages in that it is a stack oriented language of interpreted nature[1]. It is very easy to use for prototyping small real-time systems that

[1] Forth uses vectored execution and incremental compiling to speed up the run-time execution.

do not require complex coding. It allows direct memory access. The disadvantage is that it is not modular, does not support any higher-language constructs, has basic exception handling and no provisions for schedulability analysis.

2.1.4 Concurrent Languages

Concurrent programming is the name given to programming notation and techniques for expressing potential parallelism and solving the resulting synchronization and communication problems [Ben-Ari82]. Several concurrent languages have been developed, most notably, Modula-2 and Ada.

2.1.4.1 Modula and Modula-2

Although Modula, later upgraded to Modula-2, takes many of its basic features from Pascal, it was not intended to be Pascal's successor. Modula is designed to run on a bare machine with no operating system support [Young82]. A stated aim of Modula was to allow one write programs entirely in Modula, without resorting to assembler. Modula does not have a formal standard[2] and it is viewed by some as a stepping stone toward Ada [Cooling91]. Modula is well structured with strong type checking, allows direct hardware access and synchronizes its processes via waits and signals. Nevertheless, it lacks some real-time characteristics, namely, language-level exception and error handling and provision for schedulability analysis [Halang91].

2.1.4.2 Ada

Ada is not an outgrowth of any existing programming language but its development was influenced by Pascal, ALGOL, Simula, PL/1 and to a lesser degree by FORTRAN, BASIC and COBOL [Katzan84]. Ada is a large and complex language whose importance is stressed by its mandate from the U.S. military and defense agencies. It is a typical "design-by-committee" product which includes just about every conceivable feature in a modern language. Ada is hard to learn and requires a large amount of run-time support which makes it difficult to produce efficient code. Furthermore, Ada makes few provisions for schedulability analysis [Halang91].

2.1.5 Non Real-time Languages

When Kernighan and Ritchie designed C in 1972 their purpose was to give the programmer a lot of flexibility at both the higher and lower level by using higher level language constructs. C is rapidly

---

[2] However, a BSI/ISO standard is in its final approving stages [Cooling,91].

becoming one of the most important and popular programming languages [Cooling91]. C's popularity is due to its efficiency, compactness, expressiveness, modern control structures, access to low level, absence of restrictions[3], advanced development environments and availability of design tools [Cooling91], [Harbison84], [Kernighan88].

Stroustrup's C++ is an object-oriented language that is based the C language. There is no doubt that C++ will become a major force in certain programming areas [Cooling91]. Major computer manufacturers (IBM, Apple) have made serious commitments to C++ for designing future operating systems. RTC++ is an attempt to add real-time features to C++ [Ishikawa90].

2.1.6 Survey Summary

Even though not every possible language has been surveyed, the evaluation demonstrates that none of the languages give any provisions for schedulability analysis (except Real-time Euclid that will be described later). They do not provide any means for determining the execution time of the source code, or limiting its execution to a certain, pre-specified amount of time. There are two ways a programmer can be assured of the time bounds: (1) subject the code to operate under real conditions and measure the running time with hi-tech electronic equipment, or (2) use one of the timing tools described in section 2.2.

Unfortunately, there is not one "best" real-time language and, as a result, the choice has been narrowed to a handful of forerunners - C, Ada, PEARL and Forth.

2.2 Time Evaluation and Design Tools

Due to the nature of real-time systems, programmers in this area always needed more hardware and software support than their colleagues in traditional sequential programming. Many tools have been developed over the years specifically to support this task - debuggers, monitors, counters, event stream loggers, performance analyzers, profilers and simulators - the last two being of most significance for real-time.

The conventional tools are not effective for testing and timing of the time-critical part of the code because they exhibit an inherent weakness, analogous to the *Heisenberg uncertainty principle* in physics, that by attempting to measure a phenomenon one changes the very nature of the phenomenon [Yourdon72]. Each tool or service, by using some portion of the CPU time or other system resource (e.g. memory), can influence or even disrupt the internal balance of the system. For real-time systems this disturbance means that the timing of execution and response times can be significantly affected by the timing tool itself.

---

[3] C supports type definitions, but they are left to the programmer's discretion and thus are not an impediment for an experienced programmer.

Next two sections will cover two tools that require programs to be run in order to determine the execution time. The last group, presented in the last section, works at source level, gathering timing information directly from the source code.

2.2.1 Profilers

A profiler is a software tool that records how much time each function or procedure takes to execute. Depending on its design, profiler calls are inserted into all or selected functions. During execution, whenever a profiler call is encountered, a local hardware timer or a timing service is called. Information about the calling function and duration of execution is recorded into the memory block reserved by the profiler before program execution [Horspool86]. The timing units vary for each implementation with resolution in order of milliseconds on a typical UNIX systems down to 1 microsecond when hardware timers are available on a single user system.

On a single user system the profiler can record how many times each procedure was called, the percentage of profiling period spent in the procedure and minimum, maximum and average time spent in each profiled procedure. The summary is processed and displayed after the program completes its execution.

Although profilers are useful tools for identifying software bottlenecks their interference with the program execution flow can be objectionable in some applications. The more profiling data is gathered and saved, the more overhead is generated. The *trace* option (available through some profilers) generates a call-tree of procedure calls which can be especially overwhelming for most programs [MIPS89].

Further difficulties arise with profiling programs on time-sharing systems where time-slicing makes time measurement particularly complicated. Profiling then has to resort to statistical distribution of samples taken during execution. This means that in order to represent distribution of sampled data correctly, the program must run long enough. However, if sampling is done too often the interruptions to program flow will either overwhelm the program or defeat the purpose of profiling. To alleviate the situations some simplifying assumptions, such as calls to a specific routine require the same amount of time to execute, are usually made.

To test the program properly under different conditions, all or most important combinations of the whole range of input parameters must be profiled. This process in itself can be a difficult and time consuming task.

2.2.2 Simulation

For at least 25 years, simulators have been the most powerful analytical tools available. A simulation program enables a user to test and debug programs on a computer other than the target machine [Liu91]. It is a program (a mathematical model) which imitates the actions of the real system (a physical model), usually a new design or a part of the new design [Head64]. The simulation program is written to run on an existing computer and simulates generation of interrupts and performs Input/Output functions in real-time. Special packages such as SCERT or languages designed for simulation (SIMULA, SIMSCRIPT, GPSS III, SPL) are usually used [Yourdon72].

A typical input to the simulation program might include the size and speed of the proposed CPU, memory allocation, layout and speed of the communication paths, loading peaks, priorities, etc. Simulation programs are not only able to reproduce the sequence of events expected in real systems, but also they can scrutinize normal, as well as abnormal, behavior of any part of the system [Burns90].

Design usually becomes an interactive and iterative process. The process allows both the user and the simulation program to cooperate and share the task of performing the simulation [Buxton68]. Repetition of simulations with different input parameters results in a redesign effort or revision of components. The new configuration is submitted for further simulation and the process is repeated until satisfactory results are achieved.

As wonderful as this approach looks, there are formidable practical drawbacks — simulation is simply too expensive! Simulation, of all but very simple systems, is a non-trivial process requiring a team of specialists, expensive hardware and software resources[4] [Burns90]. Since simulators are real-time systems, they themselves must be thoroughly tested. Input data must be presented in the form required for the simulation to avoid the well known axiom "garbage in, garbage out". High hopes and faith that the real-time researchers have been putting into simulation are slowly disappearing — simulation programs just are not very useful when it comes to real-time [Ganssle92].

2.2.3 Source-level Time Analyzing Tools

Real-time research of the last decade has been focusing on improving performance of systems and finding methods which would guarantee performance requested by the design specifications. This need came as a direct consequence of the inability of conventional methods and tools to provide such assistance for the real-time designer.

---

[4] The NASA shuttle project simulators cost more than the real-time software itself. However, the money turned out to be well spent - many system errors were found during simulations [Burns,90].

Real-time software must be guaranteed to meet its timing constraints. Stoyenko[1987], Klingerman[1989], Halang[1991] believe that a real-time language must be designed such that its programs can be schedulability analyzable. The result of this research effort is a new experimental real-time system and a language called *Real-time Euclid* . This language provides most features that are required from real-time languages: security, concurrency, modularity, hardware access, maintainability, efficiency, and most of all, schedulability.

The knowledge about the maximum execution time (*MAXT*) is of utmost importance in real time systems. A brief introduction to the subject, definition of MAXT as well as problems associated with the MAXT calculation are discussed by Puschner[1989]. The *MAintainable Real-time System* (*MARS*), developed by a research group at the Technical University in Vienna, is a real-time design environment based on the MAXT calculation.

A *Source-Level Timing Schema* (*SLTS*), based on a belief that the software timing properties should be specified and verified at the source-level rather than at the assembly or machine levels, has been developed at the University of Washington [Shaw89, Park91].

2.2.3.1 Details of RT Euclid, MARS-C and SLTS

All three above approaches use the source level code and a compiler, or part of a compiler, to determine the timing information. All approaches require that a certain limitation be imposed on bounding loops, recursion, pointers to functions and the GOTO statements.

Real-time Euclid is a language designed specifically to accommodate schedulability analysis by using special real-time constructs such as the *noLongerThan* statement. The schedulability analyzer determines the process frames from the token file produced by the compiler.

MARS-C uses special language timing constructs (language extensions) called markers, scopes and loop sequences. They are embedded in the MARS-C programming language. The calculation of the MAXT is done in three phases. First stage saves information about the program structure in an intermediate file by parsing the pre-compiled MARS-C source code. An intermediate file containing information about program structure is created along with a C source code. Then a C compiler is run that in turn creates an assembly file. The second part aligns the saved constructs with the assembly instructions, calculates the MAXT as the sum of partial MAXT's of all primitives, sequences, alternatives, loops and subroutines and saves the data in another intermediate file. The third stage analyzes the timing data.

SLTS is based on the time prediction derived from the deterministic bounds of execution times of elementary expressions (called atomic blocks), control structures, statements and procedures. The timing tool consists of a preprocessor and a language analyzer. The preprocessor interprets user commands and prepares compiled assembly code with atomic block markers of the source program. The language analyzer includes the parser of the GNU C compiler modified by adding several lines of code which allows the parser of the timing tool to mark the atomic blocks. When an atomic block is identified the tool predicts what the object code will most likely be, and the corresponding execution time is looked up in the target machine timing table. Even though SLTS exists on its own, it can be used to predict the timing information only for the compiler and the compiler settings that the atomic blocks have been calibrated for.

2.2.3.2 Comparison of RT Euclid, MARS-C and SLTS

Park's and Shaw's schema has some, arguably considerable, language limitations such as the lack of the floating point calculations and the lack of support for important language constructs (such as FOR, DO, SWITCH). Even though RT Euclid excludes floating point operations (MARS-C does not deal with this issue), RT Euclid can be extended to support the floating point operations once a floating co-processor is added into the hardware and compiler is extended. SLTS, on the other hand, is inherently restricted in its design to allow any operations that would require library calls. Addition of an FPU into SLTS will result in more complicated prediction strategy that will lead to larger range of errors.

MARS-C loop markers add extra overhead in order of 2%. Even though SLTS uses WHILE loops, the actual interpretation of this loop resembles the FOR loop since all loops have to be bound during the analysis step. Both RT Euclid and MARS-C system accept only bounded loops. While RT Euclid bounds loops only to a number of iterations, MARS-C allows iteration or time bounding.

RT Euclid and MARS-C do not allow recursive function calls. Since the authors of SLTS do not clarify this issue, it can be assumed that SLTS cannot deal with the recursion either. Examples given by Park[1991] demonstrate only non-recursive functions.

Only RT Euclid can determine timing values of an optimized code (provided the compiler supports optimization). SLTS and MARS-C are inherently limited to consider any global optimization. For example, SLTS cannot predict atomic block content of potentially optimized code, and MARS-C might not be able to align generated code with the source-level markers, especially if some constructs are removed, their conventional structure changed, or if instructions are re-organized. For the same reasons, MARS-C could not be used with RISC architectures that rearrange the code in order to suit RISC code generation strategy.

RT Euclid and MARS-C are proprietary systems, still under development, and this significantly limits their accessibility for experimental and academic research. Moreover, MARS-C can be used only with compilers that generate an assembly language file as an intermediate output.

Chapter 3

Timing Tool Basics

This chapter will describe the basic approach of SLOTT, its timing methodology, its features and restrictions and steps that are involved in predicting the execution time. All source code examples will refer to the C language constructs.

3.1 Methodology of the Timing Tool

After reviewing the tools, we have concluded that it is important the timing methodology is language/compiler independent; the tool is easily modifiable for other hardware, generates timing information that is precise and accurate and reflects the generated code. The tool should not try to predict the execution time by guessing the generated code. It should allow precise alignment between constructs and corresponding instructions and should not exists as a preprocessor and post-processor to the compiler like in the MARS-C approach to avoid any misalignment problems. Knowing all these requirements and essentials, it is easy to conclude that the tool has to be implemented as a part of the compiler. In fact it has to be an integral part of it. The implementation should be similar to that of Stoyenko's[1987] approach in which the timing data gathering part is designed as one of many compiler's units.

3.1.1 General Aspects

Since the tool is being implemented as a part of the compiler, its implementation should be transparent to the compiler itself, *i.e.*, the tool should not hamper or even alter in any way the compilation process. The tool can just observe the compiler-specific data without changing them in any way, while the tool-specific data should be separate in their scope, context and meaning from the compiler data. However, in order to

communicate with the compiler, the tool must use the same types and structures the compiler defines and uses. Among other requirements, the tool's presence should not force the user to restrict his choice in picking out proper constructs from the language's selection. In other words, the user should be allowed to use all the language's constructs. It is only the time critical parts of the code that need most attention from the user. Here the programming style should be limited to only those options supplied by the tool if the timing value needs to be determined.

Figure 3.1 demonstrates the basic layout of SLOTT and its integration into the compiler. As the figure indicates, SLOTT does not constrain the compiler's actions in any way, rather it expands the compilers functionality. Notice that SLOTT produces output independent of the compiler's output — the *intermediate timing file*. This file is later used by the second, and compiler independent, part to generate the timing output. Also, if some of the loops are not bound after the compilation process, the analyzer will either ask the user for the bounds, or produce a formula that describes the timing.

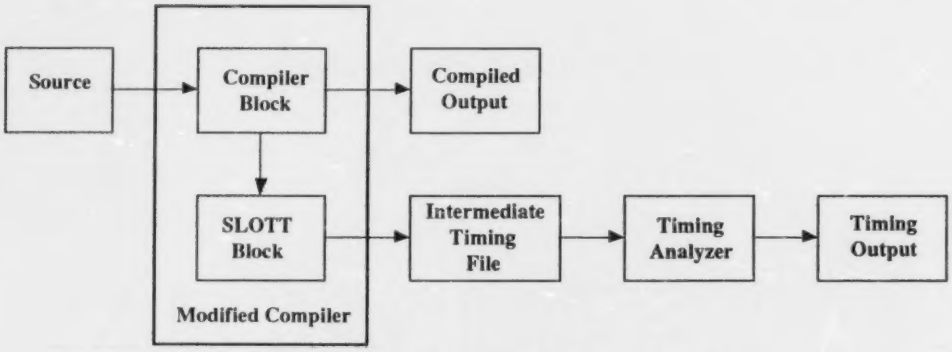


Figure 3.1 – Incorporation of SLOTT into a compiler.

3.1.2 Program Markers

The timing tool provides markers, much like in MARS-C, that direct program flow and set up iteration bounds on loops and other structures directly by the programmer within the source code. This approach is more user-friendly than the timing schema approach where the limits have to be defined each time the tool is run. Contrary to MARS-C markers, these markers are defined in such a way that they do not cause any problems when compiled on a compiler that does not have this tool built in, *i.e.*, if the markers are not recognized by the compiler, they are simply ignored during the compilation process.

These markers in C are provided inside `#pragma` statements. By definition, anything declared inside the `#pragma` statement is assumed to be compiler specific. Thus if the compiler can recognize the meaning of the `#pragma` contents, it will perform appropriate action; if the meaning is unknown to the compiler, it will skip the whole `#pragma` statement.

The implementation of these markers resembles a procedure call with the values specified in parentheses. Both strings and numbers are passed without any delimiting characters, and there can be more than one marker specified within one `#pragma` statement; these, though, have to be separated by a comma or a semicolon. The placement of `#pragma` statement is very important since the information of the markers refers to the structure that is being defined next. The markers are not case sensitive, thus `max_ic` is the same as `Max_IC`.

3.1.3 Timing Blocks

A timing block is a structure that collects the timing information. The length of a timing block can range from one line to a whole function definition. A timing block is defined by two markers – `stiming` and `etiming` – used in conjunction with `#pragma` command. The first marker creates it, the second one ends it. Timing blocks can be nested but they cannot overlap each other. The timing block can be ended only on the same nesting level it has been defined. SLOTT guards against accidental misplacement or lack of the `etiming` marker. If such a situation occurs, SLOTT will either ignore the marker or finish the timing block when the nesting level terminates, respectively. Therefore, it is acceptable to omit the `etiming` marker for the timing block that spans a whole function, or lasts until the end of the nesting level on which it is defined.

Another approach to define a timing block is to use a newly defined reserved word, like STIMING. This method insures that the compiler will keep the proper nesting level for each timing block and will disallow overlapping definitions, *i.e.*, each timing block will be treated as a statement or a compound statement. Ergo, it is not necessary to define a reserved word for ending the timing block since the block boundaries are delimited by symbols that are used for bracketing compound statements. Despite its structural approach, this technique renders itself incompatible with the language standard. Consequently, to compile a SLOTT code on a non-SLOTT compiler will require commenting out or deleting all the occurrences of the STIMING reserved word. In addition, if a timing block needs to span a whole function, most compilers have to be informed of the start of the timing block *before* the function definition. Therefore, reserved word approach will not be able to time out the whole function, just the function body.

3.1.4 Bounding of Looping Constructs and Recursive Function Calls

3.1.4.1 The Need for Loop Bounding

All tools described in section 2.2.3 consider iteration-bound loops. The Shaw's and Park's WHILE loops are bounded by the maximum number of iterations — these numbers have to be interactively entered during the timing tool execution [Park91]. Stoyenko's analyzer has only FOR loops that must be iteration-

bound before the compilation phase [Stoyenko87], and Koza's and Puschner's MARS-C system uses iteration or time bounds for their loops [Puschner89]. Loop bounding is appropriate for real-time procedures since one must know beforehand what the maximum iteration count or time will be. However, this should not be the only option left for the programmer. Thus in our approach, if this upper bound is not specified, the user will be later asked for the value during the evaluation/analysis phase. This option might even be favorable in some cases since the user can omit the loop iteration bound in order to study the "what if" situations later during the time analysis phase.

3.1.4.2 SLOTT Bounding Markers

Loop markers allow a programmer to define the upper and the total (or exact) bounds of any structures inside loop(s) or loops themselves. There are two markers that bound the loops: `max_ic` and `total_ic`, both taking a number that specifies the upper bound and total iteration count for the given structure that immediately follows the `#pragma` statement.

There is also a difference in the usage of `max_ic` and `total_ic` markers. This difference is defined not for SLOTT compiler part but for the evaluation tool which will reconstruct the execution time from the timing data. The `max_ic` marker sets the upper bound for loops; the `total_ic` marker specifies the total (exact) iteration loop count that is defined by the current and outside loops. Thus a loop that is marked by the `max_ic` will have its bound updated by the bounds of each outside loop. On the other hand, loops bounded by the `total_ic` marker will be treated as a unit outside the loops they are defined in. As an example consider the following source fragment:

```
for (i=1; i<=100; i++) {        /*  Outside loop    */

#pragma   max_ic (1000)         /*  Maximum number of iterations determined
                                    empirically before the program execution */
      while (true) {            /*  WHILE loop bounded by max_ic */
         ...
      }

#pragma   total_ic (5050)
      for (j=0; j<i; j++) {     /*  FOR loop bounded by total_ic */
         ...
      }
}
```

The WHILE loop will be bounded by 100*1000 = 100,000 iterations, by both the WHILE and the outer FOR loop iteration bounds. The second FOR loop, though, is treated as a separate entity as if it did not belong to the outside FOR loop. Consequently, the iteration count will be defined as a number calculated using this simple formula:

$$a_1 + a_2 + \ldots + a_n = \sum_i a_i = ((a_1 + a_n)*n)/2$$

Therefore the iteration count is (100+1)*100/2 = 5050, and it is no longer affected by the outside loop counts. The `total_ic` marker is then used to bound loops whose iteration count is not defined as a constant value, but rather as a variable that is dependent on outside factors that can be resolved only by the programmer.

3.1.4.3 Bounding Execution Time

In some cases it is also possible to know the maximum number of repetitions a specific piece of code will be executed inside a loop. This means that the code is executed only during some loop iterations when a certain condition is satisfied. The calculated execution time will be more accurate since only those iterations that satisfy the condition will be considered. As an example, consider the following FOR loop with the IF-THEN statement in C:

```
for (i=0; i<1000; i++) {
    if (i < 500) {
        ...          /*  Some code in IF-THEN statement   */
    }
    /*  FOR loop ends here */
}
```

Without taking into account the maximum number of iterations for the IF-THEN statement we would have to use the same number for the IF-THEN statement as for the whole FOR loop; however, knowing the upper bound we can now determine the execution time of the FOR loop more accurately. In this example, the improvement is almost two fold (if we do not consider the testing and updating of variable `i`).

Currently, the iteration bound of selection structures (IF/ELSE and SWITCH) is restricted only to the `total_ic` marker. The `total_ic` defines the total iteration count the body of the structure will be used. Thus the previous example could be written with the markers in the following way:

```
for (i=0; i<1000; i++) {
#pragma   total_ic (500)
    if (i < 500) {
        ...          /*  Some code in IF-THEN statement   */
    }
    /*  FOR loop ends here */
}
```

This feature has been described only in Koza and Puschner's MARS-C timing tool [Puschner87]. The other tools do not consider it at all. Since our design already takes into account the maximum number of loop

iterations, the extension for the other constructs, like IF-THEN-ELSE and SWITCH/CASE statements, is a straightforward addition.

3.1.4.4 Recursive Function Calls

Although all aforementioned tools forbid the use of recursive function calls, this feature is important and thus it should not be eliminated from the programmer's options. It should depend on the designer to make sure the recursive function calls are bounded especially in the time critical code that is to be timed. As Koza and Puschner[1989] mentioned in their paper, it is possible to eliminate recursive function calls using iterative techniques [Darlington78]. However, this is very close to the problem of defining iteration count for loops. Thus, if one can resolve a recursive function call as a looping construct bounded by the upper limit for iterations, it is also possible to upper-bound a recursive function. Therefore, in this implementation it will be possible to specify upper bounds on directly recursive function calls. Notice that it is difficult to determine or even predict recursive nature of indirectly recursive function calls (*i.e.*, function A calls B which in turn recursively calls A).

The bounding of recursive function calls resembles loop iteration bounding. Both `max_ic` and `total_ic` markers are supported. Since C allows function calls within function definitions, it is important the recursive function not include any functions in place of its arguments. Otherwise, the bounding value will apply to the first function that is called as an argument. The following example will bound the `traverse_tree()` recursive procedure to the maximum of 100 recursive function calls. Again this value has been empirically determined be the programmer to be the maximum value. However, if this value is unknown, or is affected by other factors, then the `#pragma` statement should be left out, and the bounding should be performed during the timing analysis phase.

```
#pragma   max_ic (100)
   traverse_tree (root);      /*  Recursively traverse the binary tree*/
```

3.1.4.5 SLOTT's Automatic FOR Loop Bounding

One of the SLOTT's strong features being implemented as part of a compiler is the possibility to gather the individual data during the compilation process. This means that we can track not only all variables that are defined, but also the values that are assigned to these variables and the operations that are performed on them. As a result, it is possible to determine the iteration count of some simple FOR loops. In fact, the FOR loop structure is totally suited for automatic bounding since all the necessary operations are present in the FOR loop definition: initialization, update, and end condition of the counter variable. Since C's FOR loops

can vary in their complexity, we have to specify the conditions that must be met in order to determine the loop iteration count:

(1) Within the FOR loop definition, the counter variable has to be initialized, tested and updated only to constants.

(2) The counter variable cannot be modified within the statements of the FOR loop body.

(3) Only the basic operations are allowed for the variable update, *i.e.*, addition, subtraction, multiplication and division. However, the use of multiplication and division should be limited since the result cannot be obtained if the multiplier/divisor is negative, zero, or one; if the bounding values do not have the same sign or one of them has a zero value.

(4) Both integer and float types are supported. Using floats should be restricted since the calculated iteration count might not reflect the actual count due to the uncertain precision and rounding problems of the float types used for predicting and actual execution.

(5) The counter variable can be a part of a structure or a union.

(6) The loop's terminating condition can contain only simple relational expressions; expressions having the *AND* or the *OR* logical operator will not be evaluated. Future version may deal with more complicated expressions.

If any of (1), (2), (3) and (6) points cannot be satisfied, SLOTT will not be able to determine the loop's iteration count. `Total_ic` and `max_ic` markers override the automatic FOR loop bounding.

3.1.5 Program Flow Statements

Special attention must be given to instructions that transfer control flow of a program outside the current level. These instructions in C are BREAK, CONTINUE, RETURN and GOTO statements. Although their difference will be described later, it should be mentioned that only those control flow instructions with predictable results will be considered, *i.e.*, BREAK, CONTINUE and RETURN, in this implementation of the timing tool. This leaves the GOTO statement out since it is hard to analyze a program in which the execution path can jump all over the place. With sound programming techniques that adhere to structured programming, it is possible to avoid the GOTO statements altogether.

The BREAK statement affects loops and the SWITCH statement. It causes an unconditional jump (or exit) out of the innermost loop or SWITCH statement. The CONTINUE statement is used only in loops since it transfers the program control to the beginning of the enclosing loop. In the WHILE and DO loops, it means that the program jumps to the loop's test condition; in the FOR loop, the control passes to the update step. The RETURN statement simply exits the current function with the value that is specified in the

RETURN statement [Kernighan88a]. Section 3.3.4 provides more details about how SLOTT deals with these control statements.

3.1.6 Informing SLOTT of Program Flow

Sometimes it is obvious what selection from the IF-THEN-ELSE and the SWITCH/CASE statements has to be considered or omitted during the analysis phase. In most of these circumstances the selection applies to an error condition which under a normal situation will not happen. Thus if the programmer is allowed to eliminate or fully consider one of the conditional statements, the extent of information needed for evaluation and analysis is that much smaller. As an example consider an IF statement that checks for an error condition and if it is satisfied, the program returns without any additional code execution:

```
if (anError) return (anError);
```

Most, or all, of the time this condition should evaluate to FALSE and thus the return should not be considered in the analysis step. Having markers that specify this kind of situation will simplify the timing analysis and presentation of the results.

To specify the program flow, two markers have been implemented: `node_skip` and `node_sel`. The first one eliminates the section from being considered, thus leaving the other alternatives as the only choices for the analysis. The second one uses that segment all the time; therefore, eliminating all other possibilities from being considered during the analysis step. The selections are specified by a number starting from 1. The IF-THEN part is 1; the ELSE part is 2. For the SWITCH statement, the numbers apply to the individual CASE statements in the order they are defined.

Using the previous example:

```
#pragma   node_skip (1)
   if (anError) return (anError);
```

this one will not consider the TRUE condition of the IF statement in the time analysis step.

None of the aforementioned source-level timing tools describe this option. Even though it is not as important as the possibility of bounding loops, its usefulness becomes obvious in fine-tuning the exact program flow, thus eliminating unnecessary information and calculating more precisely the execution time.

3.1.7 Inline Assembly

In some situations it is necessary to use the inline assembly language to produce more compact and faster code. This is particularly required in real-time systems that try to utilize every feature of their hardware. Thus if the hardware contains an FPU, the programs should be able to exploit the FPU instructions for all floating point operations rather than to use library calls. Some compilers though cannot produce FPU instructions for transcendental and trigonometric instructions. Consequently, inline assembly instructions must be used to compensate for this deficiency. Our implementation of SLOTT is able to register every assembly instruction – of either compiler or user origin.

This need for inline assembly has not been addressed in any of the source-level timing tools. However, this feature might be adopted in RT Euclid and MARS-C if the compilers support the inline assembly. The timing schema on the other hand would have to be modified to directly recognize not only the C language instructions, but also assembly language instructions.

3.2 Theoretical Approach to the Execution Time Prediction

In order to predict the execution time of a function or even a part of a function, one needs to understand the structures of the high-level language. Sequential languages such as Pascal, C, Ada, Modula-2, etc. share certain basic constructs, although they may take different forms.

3.2.1 Basic Constructs of a Procedural Language

The basic constructs can be divided into these categories: simple statements, compound statements, conditional statements (or alternatives), loops and individual functions [Puschner89]. Unlike MAXT[1], a term that has been defined by [Puschner89], a similar term will refer to the *P*redicted *E*xecution *T*ime (*PET*) value. PET can refer to any user-defined option: minimum or maximum.

3.2.1.1 Simple and Compound Statements, and Functions

Simple statements are any instructions that cannot be broken down by the parser into simpler units. These can include simple constructs or expressions. The execution time of such an expression is just the sum of all CPU/FPU instructions that have to be generated to execute the expression:

$$PET(simple) = \tau(simple) \qquad [1]$$

[1] Maximum Execution Time.

where τ *(exp)* refers to the time value of all assembly instructions that make up the expression *exp*.

Compound statements contain any number of simple statements. Again the execution time is just a straightforward addition of execution time values of individual simple statements:

$$\text{PET (compound)} = \sum_i \text{PET (construct}_i\text{)}. \qquad [2]$$

A function can be divided into three sections: *epilogue*, *body* and *prologue*. The prologue sets up the local frame pointer and saves important registers while the epilogue restores the registers and the stack pointer. The body represents all instructions defined by the programmer. The execution time of the function is the execution time of the instructions that make up the three blocks:

$$\text{PET (function)} = \text{PET (prologue)} + \text{PET (body)} + \text{PET (epilogue)}. \qquad [3]$$

3.2.1.2 Conditional Statements

Conditional statements are composed of a conditional expression and a statement or a compound statement that follows the expression. In C/Pascal these constructs are represented by the IF-THEN-ELSE, and the SWITCH/CASE statements respectively.

The execution time of a conditional, the IF-THEN-ELSE statement, can be calculated in two ways. In the first case, the timing value T_{CS} is just an addition of the execution time of the conditional expression T_{cond} and one of two alternatives (T_{IF} or T_{ELSE}):

$$T_{CS} = T_{cond} + (T_{IF} \text{ or } T_{ELSE})$$

The other option assumes only the time of each alternative, leaving the conditional time to be considered outside the execution time of the whole conditional statement:

$$T_{CS} = op(T_{IF}, T_{ELSE})$$

where *op* is the max or min operator.

Since it is not important how we keep track of execution time of conditional statements, the latter approach is more compact — the execution time of any conditional statement will be expressed just by timing information of the selected branch leaving the timing information of the conditional expression to be considered outside the conditional statement. Additionally, the implementation is much simpler since the steps in gathering the timing information follow the compilation process. If the minimum/maximum execution time is desired then the time is just the min/max time value of all the alternatives.

Some compilers perform short-circuiting in the evaluation of a conditional expression. In other words if the truth value of the conditional expression can be satisfied without evaluating all sub-expressions, a jump is taken to the appropriate conditional statement. This feature leads to a faster execution. Some compilers; however, have to evaluate all conditional sub expressions. For example, if a programmer were to test a field of a dynamically allocated structure, he would have to check for the presence of that structure by examining its pointer first before any testing of that structure's field. In non-short-circuiting conditionals, this would have to be accomplished with two IF-THEN statements, while in short-circuiting versions, one conditional statement would be sufficient with two sub-expressions in the conditional expression.

Since time evaluation can consider short circuiting only if the values of all variables involved in the conditional expression are known, the non-short circuiting has to be assumed. Therefore, the timing value of the conditional will be calculated for the whole expression. This also would have to be assumed for the worst case evaluation anyway.

The PET value of the conditional is then defined as:

$$\text{PET (conditional)} = op\ (\text{PET } (\text{stmt}_1), \text{PET } (\text{stmt}_2), ..., \text{PET } (\text{stmt}_n)) \qquad [4]$$

where

op defines either maximum or minimum PET value of selection statements *1* through *n*;
PET (condition) is considered inside the construct that defines the conditional.

3.2.1.3 Loops

Loops are similar to conditional statements with an additional jump instruction that transfers program flow back to the conditional expression (WHILE, FOR loops), or to the beginning of the loop depending on the evaluated condition (DO-WHILE/REPEAT-UNTIL loops). Contrary to conditional statements, the execution time of a loop is *the sum of the time taken for the evaluation of the conditional and all statements inside the loop multiplied by the number of loop iterations*. Some loops include the initialization of variables, like the FOR loop, so these initialization instructions must be included in the overall calculation of the execution time. Even though present as the part of a loop, the initialization part has to be excluded from the loop's execution time. Rather it has to be defined as a part of the code in which the loop is defined. The following FOR loop

```
FOR (i=0, j=1, <etc.>; <conditional>; <update>)   {
    /*  Statements inside the loop*/
}
```

is the same as:

```
i = 0;
j = 1;
<etc.>
FOR (/* empty */; <conditional>; <update>) {
   /*  Statements inside the loop*/
}
```

In order to determine the execution time of a loop exactly, we must know how many iterations a loop will execute. Even though we are calculating the worst case execution time, the best case will have to include minimum loop count. The minimum loop count of both the WHILE and FOR loops is defined as zero, while the DO loop is defined as one. Thus there will be two definitions for loops; one for WHILE and FOR loops:

$$PET(loop) = PET(condition) + ic * (PET(condition) + PET(loop_body)) \quad [5]$$

and the other for a DO loop:

$$PET(loop_{DO}) = ic * (PET(condition) + PET(loop_body)) \quad [6]$$

where

ic is the loop's iteration count.

3.2.1.4 Execution Time Constructs Summary

This summary is presented in a Table 3.1 that describes the execution time of all the previously described constructs in their mathematical form.

Construct/ Expression ,	Predicted Execution Time (PET)
simple	$PET(simple) = \tau(simple)$
compound	$PET(compound) = \sum_i PET(construct_i)$
conditional	$PET(conditional) = [PET(condition)]^{\dagger} + op(PET(stmt_1), pet(stmt_2),...,PET(stmt_n))$
DO loop	$PET(loop_{DO}) = count * (PET(condition) + PET(loop_body))^{\ddagger}$
FOR/WHILE loop	$PET(loop) = PET(condition) + count * (PET(condition) + PET(loop_body))^{\ddagger}$
function	PET (function) = PET (prologue) + PET (body) + PET (epilogue)

Table 3.1 - Constructs of a procedural language and their timing formulas.

Notes:

τ(exp) is the time of individual assembly instructions generated for *exp*.
op is either minimum or maximum operator.
$stmt_1,..., stmt_n$ is a construct 1 through n used as a choice from all options in the conditional.
† the execution time of this conditional is added to the construct in which the current conditional is defined and thus it is not considered as part of the conditional.
‡ the iteration count for DO loops is defined to start from 1; other loop types start from 0.

3.3 Timing Process

The timing process is identical to the Real-time Euclid and MARS-C system, where both systems record execution time of each instruction that is produced by the compiler. However, the two systems are different in their approach to gather the timing information. Our approach resembles the Real-time Euclid in that SLOTT has been implemented as part of a compiler. Before engaging ourselves in the process description, we have to introduce structures that are used for recording timing information — *timing blocks* and *timing nodes*.

3.3.1 Structure of Timing Blocks and Nodes

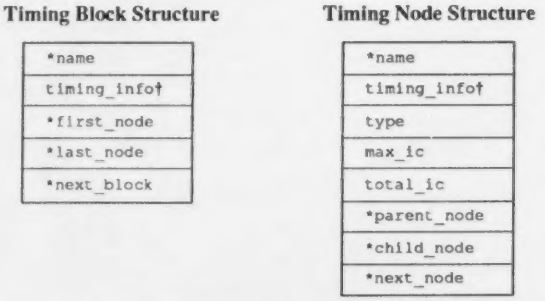
A *timing block* describes a chunk of code that the user wants to time. The code length that the block can record spans from one instruction to the whole function. Each block is defined by two `#pragma` markers `stiming` and `etiming` (or a reserved word STIMING). Hence, each block knows its starting point, ending point, name and what constructs belong to the block. More than one block can be defined for one function; however, they have to be nested, *i.e.*, they cannot overlap. The reason for restricting the blocks to resemble nesting of constructs is to prevent ambiguities such as this one:

```
#pragma   stiming (time_blk)  /* Start timing block outside IF statement */
...
if (cond) {
   ...
#pragma   etiming (time_blk)  /* Should end outside the IF statement  */

   }
   else   {
   ...
   }
```

where timing is defined for the IF-THEN statement but not for the ELSE statement! SLOTT makes sure that bounding markers, for the timing block, are defined on the same level. The block's name is specified with the `stiming` marker; it does not have to be used with the `etiming` marker. If the `stiming` marker is specified before the function definition, the whole function is timed. In this case the block's name does not have to be specified with the `stiming` marker if the block's and function's names are intended to be the same. Since timing block definitions can span at most one function, the `etiming` marker is not needed at the end of the function definition because SLOTT automatically terminates all unfinished timing blocks. As mentioned in section 3.1.3, the `etiming` marker can be left out whenever the timing block spans the whole nesting level.

The *timing node* structure maintains the timing information that is specific to one construct — either a loop, conditional (selection) statement or a function call. Each timing node holds timing information only for those statements that are present inside the corresponding construct and that can be represented as simple expressions (table 3.1). This means that if the construct contains other complex expressions, *i.e.*, other constructs, then each of them will be assigned a new timing node which will contain timing information specific to that construct. Besides the timing information, the timing node also contains fields for accessing other nodes on the same level, child level and parent level. If a timing block contains one or more timing nodes, the first and last one are recorded in the proper fields of the timing block (Figure 3.2).



† The size of the timing_info fields depends on the CPU/FPU configuration.

Figure 3.2 - Timing structures used by SLOTT.

Each timing block can access all the constructs defined in the block through its two fields — the `first_node` and the `last_node`. These fields point at the timing nodes of the *first* and *last* constructs defined on the same nesting level as the timing block. These fields are NIL if no construct has been encountered so far during the compilation process. If there is only one construct, both fields contain the same address value of the corresponding node.

The `next_node` field of a timing node points at the following node on the *same level*, and the `child_node` points at the first construct that is defined in the current construct. Each node can point only at the first child; however, every child points at its parent node through the `parent_node` field. Thus the rest of the children can be accessed through the first child and the `next_node` pointers of the child's siblings.

Both the timing block and the node contain fields for storing their names. The block's name corresponds to the symbol that has been defined with the `stiming` marker at the beginning of the block's definition.

The node's name describes the current construct and all constructs that embody it, *i.e.*, the name is built up from construct abbreviations and numbers. Such numbers indicate the order of the construct's occurrence

on the level it is defined. To demonstrate the formation of the nodes' names, consider the skeleton of the a C function `func()`:

```
func ()
{
    do {                        /* DO-WHILE 1 */
    ...
    } while ();

    if () {                     /*  IF 2      */
        ...
        while () {              /*  WHILE 1   */
            ...
        }

        for () {                /*  FOR 2     */
            ...
            if ()               /*  IF 1      */
                continue;
            ...
            switch () {         /*  SWITCH 2  */
            case1:              /*  CASE 1    */
                ...
            case2:              /*  CASE 2    */
                ...
                break;
            case3:              /*  CASE 3    */
                ...
                return;
            }
        }
    }

    else   {                    /*  ELSE 3    */
        ...
    }
}.
```

The numbers in the comment fields represent the sequence of constructs defined on the same level. Thus IF 1 is the first construct defined in the FOR loop, and SWITCH 2 is the second one. All CASE statements belong to one SWITCH statement and their numbering starts from 1. The name of each construct is the concatenation of individual outer constructs starting from the outermost one. For our example, the names will be defined as follows (listed in the order of appearance): *d1*, *i2*, *i2w1*, *i2f2*, *i2f2i1*, *i2f2s2*, *i2f2s2c1*, *i2f2s2c2*, *i2f2s2c3*, *e3*; where *d*, *w* and *f* stand for DO, WHILE and FOR loops respectively; *i*, *e*, *s* and *c* stand for IF, ELSE, SWITCH and CASE statements. The nesting level is maintained for all constructs during the compilation process even if no timing block has been defined.

3.3.2 Maintenance of Timing Structures during Timing Process

Before the dynamics of the timing process, along with the creation and maintenance of timing structures, are defined, several global variables must be defined: the `ts_ptr`, `fts_ptr` and `cur_node`. These

variables are pointers that support the timing blocks and nodes. `ts_ptr` and `fts_ptr` are the timing block pointers; the first one points at the link list of timing blocks whose constructs are just being parsed through. The second one points to all those timing blocks whose constructs have been finished. Each new timing block is inserted into the link list such that the `ts_ptr` points to the newly created block, *i.e.*, the pointer always points to the timing block that is the most current one. This scheme allows effective re-linking because we do not have to be concerned with overlapping timing blocks that would require more elaborate design. This approach resembles FIFO method of the stack that is suited for maintaining nested constructs. When the construct terminates, its timing block is re-linked in reverse order from the `ts_ptr` link list and inserted into the link list pointed to by the `fts_ptr` pointer.

The `cur_node` pointer points at the timing node whose corresponding construct is being traced through. If the generated instructions are on the same level as the outermost timing block definition, the `cur_node` pointer is initialized to NIL. If the construct finishes, the `cur_node` is re-linked to point at the parent node. This re-linking is accomplished by the `parent_node` fields of the timing node structures.

3.3.3 Steps of the Timing Process

The timing process consists of the following steps, not necessarily executed in the numerical order:

(1) For each timing block allocate the timing block structure and insert it into the link list pointed to by the `ts_ptr` pointer.

(2) Recognize each construct, allocate a timing node for it and update the `cur_node` pointer.

(3) Record the execution time of each instruction in (i) the timing node pointed to by the `cur_node` if the pointer is not NIL; and in (ii) any timing blocks whose nesting level corresponds to the current parsing level (*i.e.*, the timing blocks that do not have any nested constructs at the current parsing stage).

(4) As constructs terminate, the `cur_node` pointer is re-linked from the current node to its parent node. If the parent node, thus `cur_node`, becomes NIL, then the timing information is recorded only in the timing block structure (corresponds to the situation of step 3ii).

(5) When the timing block terminates, re-link its pointer from the `ts_ptr` to the `fts_ptr` link list.

(6) At the end of the function definition, terminate all timing blocks, write out the timing information into the intermediate file, deallocate all structures and re-initialize the global variables.

To illustrate this process, consider the following C code that outlines construct declarations in `fnc()` function and the accompanying Figure 3.3 that shows how these structures are created and maintained (the letters in the comment fields correspond to the figure parts):

```
                                    /* a   */
#pragma   stiming (fnc)             /* b   */

fnc () {

   ...
   if () {                          /* c   */
      ...
   }                                /* d   */

   for   {                          /* e   */
      ...
#pragma   stiming (if_else)         /* f   */
      if () {                       /* g   */

         ...
      }
      else   {                      /* h   */
         ...
      }
#pragma etiming ()                  /* i   */
   }                                /* j   */
}
                                    /* k   */
```

The following are the comments that apply to the `fnc()` function:

(a) Before the function is parsed, all global variables are initialized to NIL.

(b) As soon as the `stiming` marker is encountered, a new timing block structure is allocated and the `ts_ptr` is set to point at it. The name of the block is set to *fnc*. This layout will last throughout the function definition until the first construct is encountered. During this stage the timing information is recorded inside the *fnc* timing block.

(c) This step marks the first occurrence of a construct where a timing node is allocated. Both the `first_node` and the `last_node` along with the `cur_node` are set to point at the newly created node.

(d) Once the IF statement is finished, the `cur_node` is linked to the `parent_node` of the *i1* node, which in this case is NIL. Hence, all timing information is again recorded in the *fnc* block.

(e) Another timing block is allocated for the FOR loop construct. The `cur_node`, `next_node` of the *i1* timing node and the `last_node` of the *fnc* block are all updated to point at the new *f2* node.

(f) This stage encounters another `stiming` marker. A new timing block is allocated; it is inserted in front of the *fnc* block, and its name is set to *if_else*. The time gathering process will satisfy both conditions (i and ii) of step (3) mentioned above, *i.e.*, the information will be stored in both the node pointed to by the `cur_node` and the *if_else* timing block.

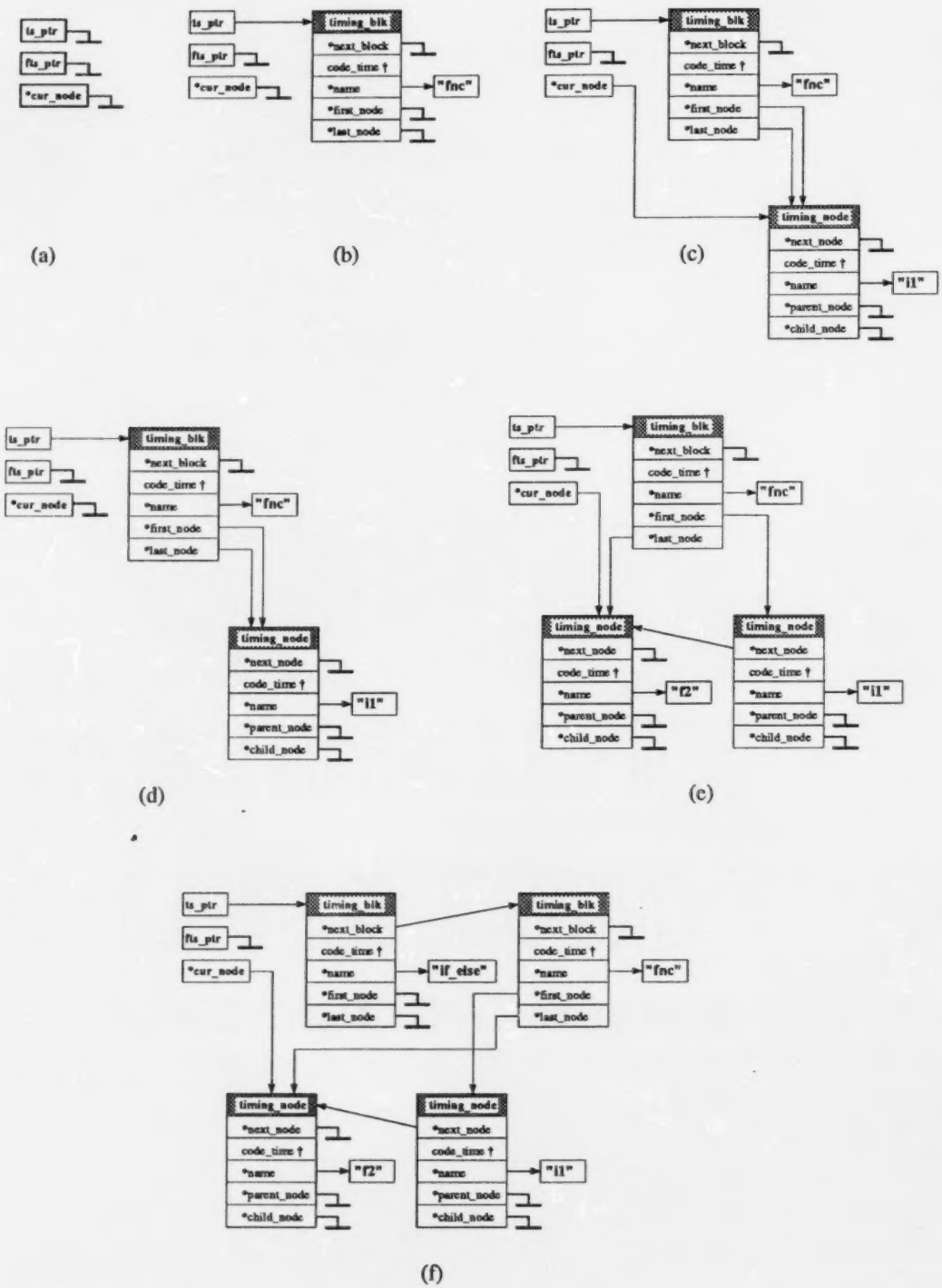
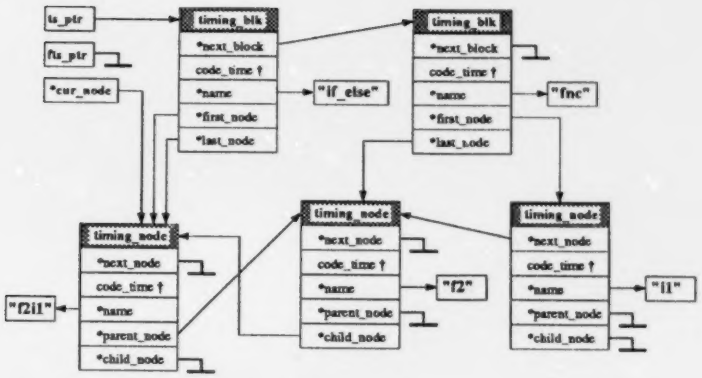
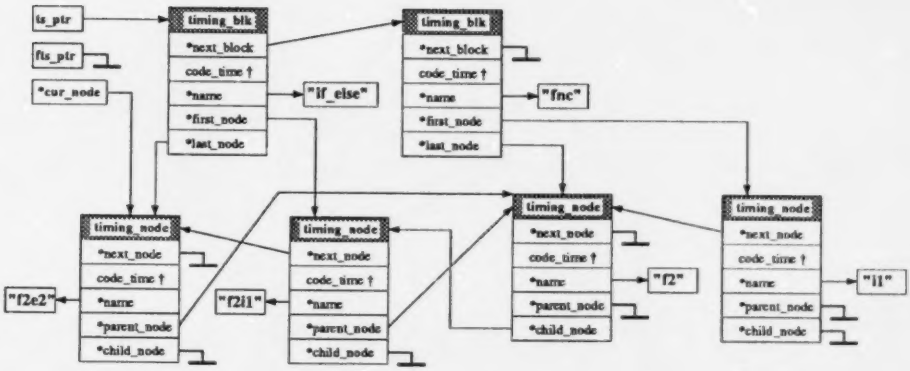


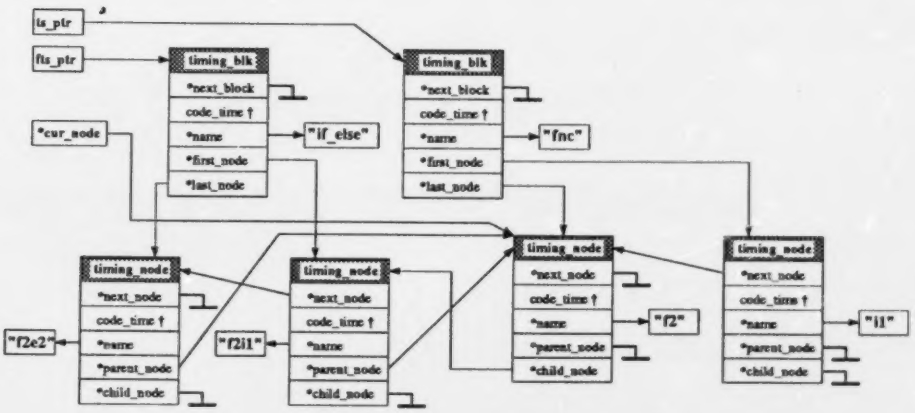
Fig 3.3 (a-f) - Creation and linking of timing blocks and nodes.



(g)

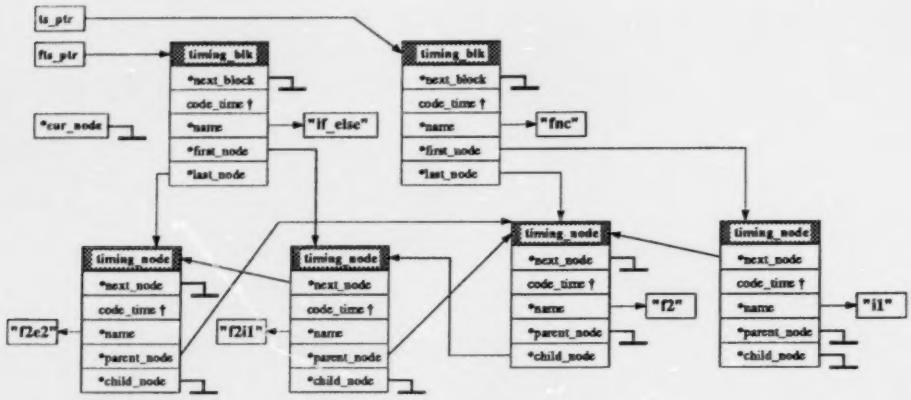


(h)

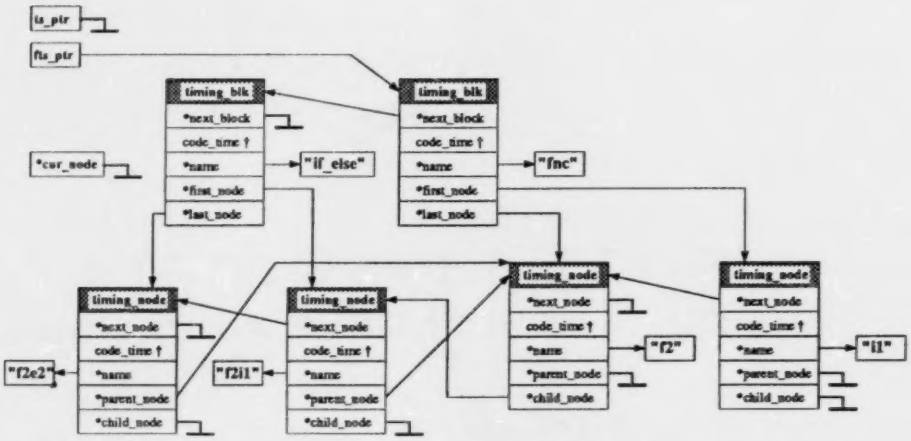


(i)

Fig 3.3 (g-i) - Creation and linking of timing blocks and nodes.



(j)



(k)

† The format and number of the timing fields depend on the hardware.

Fig 3.3 (a-k) — Creation and linking of timing blocks and nodes. Before the function is parsed, all global pointers are initialized to NIL (a). Just after the first `#pragma` statement, the first timing block - *fnc* - is created (b). For each of IF and FOR construct, a new timing node is allocated and cur_node is linked appropriately to the currently parsed construct (c, d, e). A new timing block - *if_else* - is recognized by the `#pragma` (f) and a new construct IF is inserted (g). Subsequently, the timing node structure for the ELSE construct is allocated (h). Next the `#pragma etiming` statement is encountered which will force the *if_else* timing block to be relinked from `ts_ptr` to `fts_ptr`, and `cur_node` pointer to point at FOR loop (i). Section (j) shows the part when the FOR loop is terminated and the `cur_node` is set to NIL since no other structure follows the FOR loop. The whole function finishes its definition (k) and the timing block for the whole function *fnc* is re-linked from `ts_ptr` to `fts_ptr`. At this point the timing data is collected from the timing blocks and nodes. Finally, all structures are discarded and the global pointers are reset to NIL.

(g) The IF statement brings about allocation of a new timing node *f2i1*. This time the *f2i1* is created as the child of the *f2* node. Since the *f2i1* node is defined on the same nesting level as the *if_else* block, both the `first_node` and the `last_node` of the *if_else* block will point at it. The `cur_node` is also updated.

(h) The ELSE part of the IF statement is treated in the same way as the IF-THEN part meaning a new node structure is allocated and named *f2e2*.

(i) This part ends the *if_else* timing block. The corresponding structure is re-linked from `ts_ptr` to `fts_ptr`. Since the ELSE has already finished, the `cur_node` is updated to point to the *f2* node of the FOR loop.

(j) By finishing the FOR loop, the cur_node is set to NIL.

(k) Finally, all timing blocks are removed from the `ts_ptr` link list and placed on the `fts_ptr` list (k). The timing information is written into the intermediate timing file, all structures are freed and the global variables are initialized to NIL, *i.e.*, to resemble the first (a) state.

3.3.4 Treatment of Control Statements

When a control statement like BREAK, CONTINUE or RETURN is encountered, the time gathering process is suspended immediately without terminating any node or block. This is necessary to avoid any confusion when a code mistakenly follows the control statement. This means that SLOTT recognizes the instructions and constructs, but it does not accumulate the timing information. The only situation when SLOTT can discard any statements and constructs is after the RETURN statement that appears in the outermost layer.

3.4 Execution Time Analyzer

Once all the timing information has been assembled in the intermediate file(s), the second part of SLOTT – the execution time analyzer – can be used to determine the final execution time values of all timing blocks and present them in more meaningful and understandable forms.

3.4.1 Reasons for Compiler Independent Implementation

It might not be that obvious, but the analyzer implementation that is independent of the compiler is not only a very useful, practical and beneficial approach, but also a necessity and requirement.

The benefits include machine and compiler independent implementation, that permits analysis of programs designed for different hardware configurations. Hence, the analyzer does not have to be implemented in a specific language like C, but rather, it can be designed in any language and the timing information can even be imported into programs like databases and spreadsheets.

Being implemented as a separate unit from the compiler, the analyzer is very easy to modify and to suit any user's needs. Anybody can change the analyzer's code without actually touching any part of the compiler. Moreover, the compiler carries a huge baggage of source code that is very time-consuming and space-requiring to compile. In most cases, the source code is unavailable to the user for proprietary, licensing or other reasons.

More importantly, the reason for splitting the analyzer from the compiler is to calculate the execution time of timing blocks from the timing data. For example, a timing block can contain functions that are defined before or after the current timing block, or even in a different file. In order to determine the execution time of the block, we need to know the execution times of all the functions that are used inside the block. Consequently, we would have to suspend the calculation of the block's execution time until we gathered the timing information for all involved functions. Moreover, if any of the functions were defined in a different file and have not been encountered during the compilation, then the compiler would have to search for the corresponding function in all project files and forcibly recompile that file again in order to obtain the necessary timing data. This would require very complex compiler modifications that would lead to time-consuming recompilations of already compiled source files. By having intermediate files containing timing information, we can leave the compilation process and time gathering process for the compiler, while the calculation process of the timing blocks is left to the analyzer.

3.4.2 Analyzer's Output

The analyzer can analyze timing data for different hardware. It is informed about the hardware configuration from the intermediate file in order to decide whether it can deal with the specified hardware and to know how many fields are defined to store the timing information.

If all timing blocks are bounded, *i.e.*, we can calculate execution time of every construct defined in the blocks, then the analyzer's output for each block is a CPU/FPU count along with the time in seconds that is derived from the corresponding count and frequency. However, if some of the constructs are not or cannot be bound, then the analyzer can (1) iteratively ask the user about the bounds and thus produce immediate answers (the "what if" situation analysis); or it can (2) generate a formula that will reflect the execution time as a function of the unbound variables (iteration count).

To illustrate the analyzer's output when the loops are not bound, consider the following `SimpleStrMatch()` procedure similar to the one that has been used in our evaluation process:

```
#pragma stiming()
SimpleStrMatch (tstr, pstr, tlen, plen)
char   *tstr, *pstr;
int    tlen, plen;
{
    int    i, j, k;

    i = j = k = 0;

    while (j < tlen && k < plen)  {

        if (tstr[j] == pstr[k]) {
            j++;
            k++;
        }
        else   {
            i++;
            j = i;
            k = 0;
        }
    }
    if (k > plen) return i;
    else   return j;
}
```

Since the while loop is not bound, the analyzer cannot determine the total number of CPU cycles. Instead, it can produce the following output, that describes the timing information of the `SimpleStrMatch` time block in terms of three functions. Each function represents a formula that describes execution time in terms of CPU instruction timing table used and all the constructs involved in a timing block. Since MC68020 data book gives three different timing values for every instruction – *best*, *cache* and *worst* – we have three functions to represent all three cases.

```
Program Structure and its variables:
    WHILE: n1
      IF:  b1
      ELSE:    b2
    IF:    b3
    ELSE:  b4

best:  41 + n1(42 + ([b1*23] or [b2*17])) + ([b3*6] or [b4*6])
cache: 59 + n1(92 + ([b1*24] or [b2*23])) + ([b3*13] or [b4*13])
worst: 80 + n1(122 + ([b1*33] or [b2*34])) + ([b3*18] or [b4*18])
```

where `n1` is the iteration count of the WHILE loop; `b1` through `b4` are Boolean variables having value 0 or 1. The selection of the Booleans is reflected by having one Boolean variable equal to 1, while the other Boolean variables in the same expression are equal to 0.

Given the WHILE loop iteration count, we can calculate the number of CPU cycles for the best and worst program paths. The worst path of a program is obtained by choosing a selection of the

IF/ELSE/SWITCH statements with the most CPU cycles. If the loop is bound to 500 iterations then the *worst* case count of the worst path is (*i.e.*, `n1=500, b1=0, b2=1, b3=1, b4=0`):

$$80 + 500(122 + 34) + 18 = 78098 \text{ CPU cycles}$$

which is exactly the same number when `max_ic()` marker is used to bound the loop to 500 iterations.

Chapter 4

Implementation of SLOTT

SLOTT consists of two separate parts – one is implemented as the integral part of a compiler, the other as an analyzer. The compiler part can be further divided into two parts – the compiler specific part and the tool specific part. This discussion will concentrate on both compiler parts. First, the modifications that need to be done on a C compiler will be described. Then more details about the changes that are compiler specific will be presented. More precisely, particular aspects that pertain to the modification of the GNU C compiler (GCC) will be given.

4.1 Compiler Modifications

As indicated before, the timing information is gathered during the compilation process. This means that the recording part of the timing information has to be implemented as part of a compiler.

Before we specifically describe modifications pertaining to certain parts of a compiler that need to be modified, we have to take a closer look at the compiler itself, its basic blocks and how they are connected to form one uniform structure — a compiler. We will try to be as non-specific as possible; however, in most cases when we present an example we will refer to C definitions and constructs. The next section will deal with the basics of a compiler and compilation process. It has been included here only to make our implementation clear.

4.1.1 Compiler Fundamentals

The compiler is a program that accepts an input program, usually written in higher-level language, and generates output in the form of another program — a low-level language such as assembly language or

machine object code. A compiler can be divided into several distinct blocks. These are *lexical analyzer*, *parser*, *intermediate code generator*, *code optimizer*, and *code generator* (Fig 4.1). Additionally, any errors are handled globally by the *error handler* and the parser itself that tracks any syntactic errors in the source program. The intermediate code generator and code optimizer are optional, since a simple compiler can be designed without them.

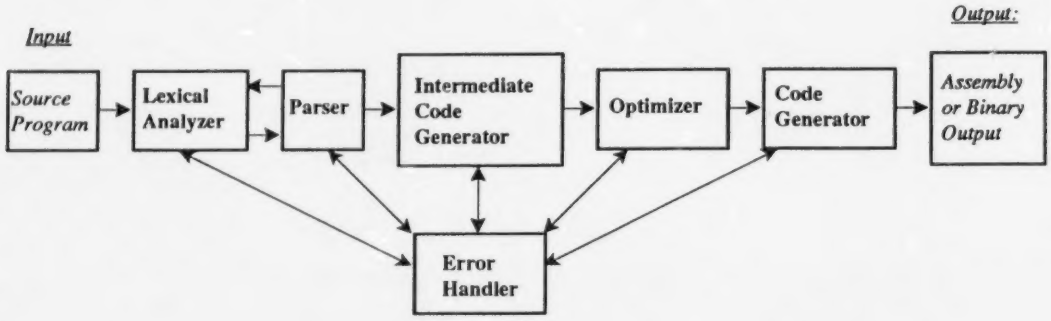


Figure 4.1 - Basic blocks of a compiler.

4.1.1.1 Compilation Process

Compilation is a complex process that cannot be described in one or two sentences. Therefore rather than giving a detailed description, we refer the reader to some of the references: [Aho72], [Aho77], [Barret79], [Bauer76], [Fisher88], [Polak81], [Pollack72]. Let us just mention a couple of points that summarize the compilation process as we introduce each compiler block. More detailed description will be presented when appropriate.

One of the main parts of a compiler is the *lexical analyzer*, or scanner, that reads the source, extracts the words and categorizes them into *tokens*. The meaning of the tokens is known to the *parser*, or *syntax analyzer*, that reconstructs the tokens into a *parse tree* (Fig 4.3) according to the grammar rules specific for the target language[1]. As the grammar rules are satisfied, the parser executes compiler specific routines that are particular either to code generation or intermediate code generation. *Single-pass* compilers produce the output code (assembly or binary) as soon as the rule is satisfied, leaving very little for code optimization. The only code optimization (done at code generation) is of a local type and accounts for the reduction of loads or stores if a known variable is present in a CPU/FPU register. However, *multi-pass* compilers employ other parts that do not exist in single-pass compilers — an *intermediate code generator* and *optimizer*. The intermediate code generator is necessary for optimization purposes and construction of multi-platform compilers. The optimization is performed over several passes, in which each pass handles a class of instructions such as jumps, loops and register allocation.

[1]These language-specific rules are encoded into the parser, so any modifications to the grammar rules must be accompanied by appropriate changes of the parser rules.

Only the parsing stage is the most common among different compilers; the other processes like intermediate language generation, optimization and code generation are very specific to each different compiler implementation. Hence the parsing process will be described in more detail than the code generation phase.

4.1.1.2 Parsing Process

Some compilers use LR parsers for compilation. These parsers scan the input from *L*eft-to-right and construct a *R*ightmost derivation in reverse [Aho77]. An LR parser consists of two parts, a *driver routine* and a list of all grammar rules or a *parsing table*. The *parser generator* constructs a parser by translating grammar rules and expresses them in terms of a parsing table and a driver routine (Fig 4.2):

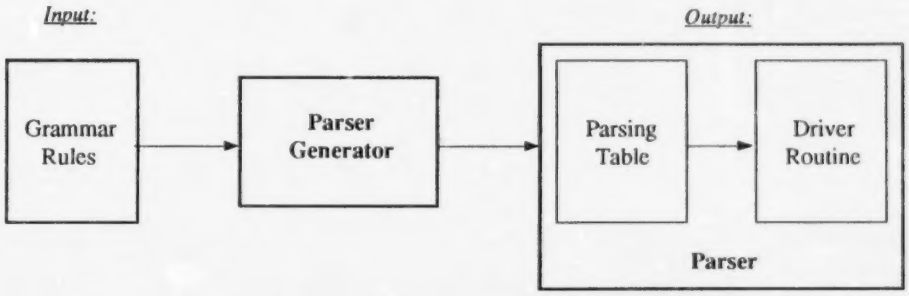


Figure 4.2 - Parser generation from the grammar rules.

As mentioned in the previous section, the parser works closely with the lexical analyzer to construct the parse tree from tokens (Fig 4.3). Each time the parser needs a new token, it asks the lexical analyzer for it. A token, which is in the form of a numeric code, represents a sequence of characters that is treated as a logical entity. Tokens consist of reserved words (IF, THEN, ELSE, etc.), identifiers, constants, operators and punctuation symbols. Once one or more tokens can be combined according to a grammar rule, compiler specific routines are invoked. Depending on the type of the compiler, these procedures either produce intermediate code or directly call code-generating routines.

Knowing the basic functions of each compiler unit, we can now concentrate on the modifications that need to be performed on compiler blocks, specifically, the parser, lexical analyzer and code generator. The following section will outline the modified compiler areas.

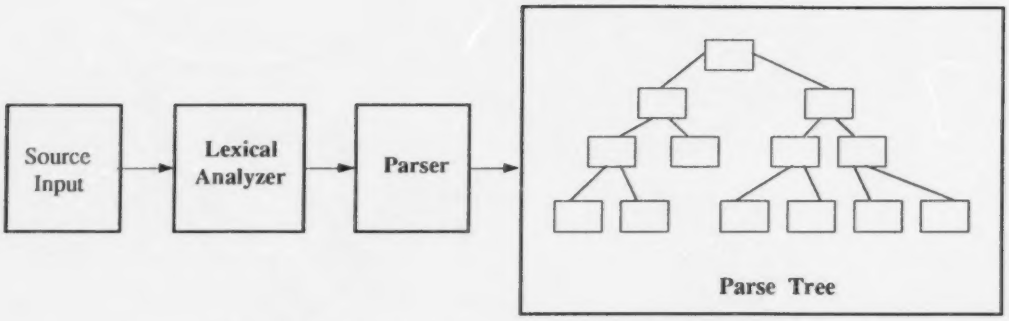


Figure 4.3 - Parser in action producing a parse tree of tokens.

4.1.2 Outline of Compiler Modifications

The main compiler blocks that have to be modified are: the lexical analyzer, parser and the code generator. Figure 4.4 shows these parts along with the machine analyzer that exists on its own. The machine analyzer is a hardware dependent part since it records the timing information of a specific hardware.

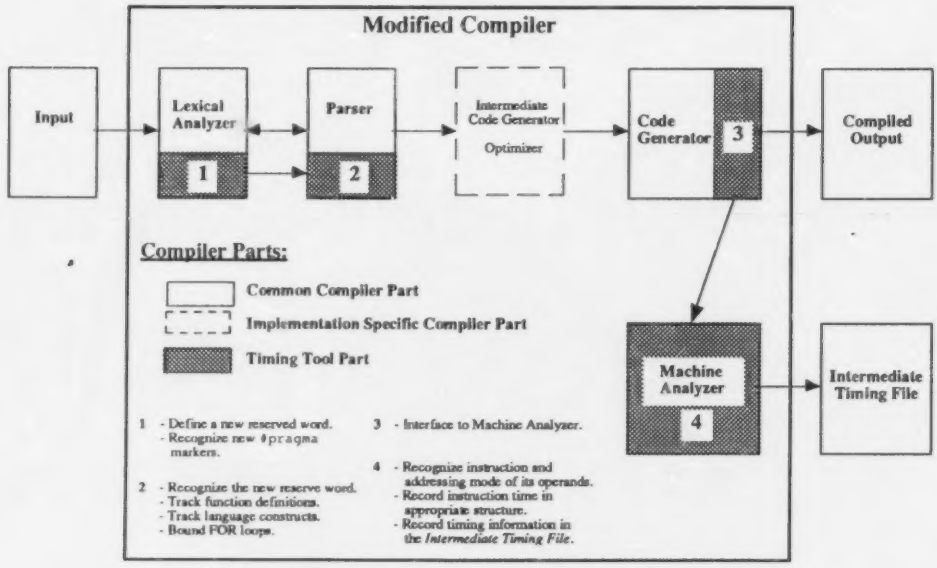


Figure 4.4 - Parts involved in the modification of a compiler.

Even though the modified parts are shown in separate blocks as being parts of the lexical analyzer, the parser and the code generator, they actually can be separated into more layers which can communicate between each other without the need to involve the compiler. As an example, the lexical analyzer's SLOTT layer communicates with the parser's counterpart without the need of using the compiler parts.

4.1.3 Lexical Analyzer Modifications

The lexical analyzer has to be modified in order to understand the new `#pragma` markers and to carry out the appropriate actions. Also, if a new reserved word approach is used to demarcate the timing blocks, then the functionality of both the lexical analyzer and the parser has to be extended.

4.1.3.1 #pragma Approach

Adding new `#pragma` markers depends largely on the lexical analyzer support for other `#pragma` markers. If the compiler already recognizes some markers, then the procedure that recognizes them has to be modified to include our new markers: `stiming`, `etiming`, `max_ic`, `total_ic`, `sel_node` and `skip_node`. Additionally, an appropriate function must be called when the marker is recognized to inform the compiler about the user value passed through the marker. If the compiler does not recognize any `#pragma` markers, then we have to insert the procedure that will identify all our markers.

4.1.3.2 Reserved Word Approach

The reserved word approach extends the parser's functionality since it allows the parser to treat each timing block as a construct. This means that the timing block can be defined by using only one word – `stiming` – much like a WHILE loop with the `while` reserved word. A timing block of any length can then be delimited in the same way as any other compound construct is delimited — by curly brackets. The advantages of using this approach are (1) no need to use the second bounding marker, (2) the parser automatically checks the nesting level of constructs. The disadvantage of using the reserved word is in generating non-standard code that can only be compiled on the compiler that recognizes the reserved word. Additionally, the reserved word approach cannot time the whole function, only the function body.

The lexical analyzer modification involves inserting the new reserved word into the list of all reserved words so that the lexical analyzer can recognize it and pass the appropriate ID number to the parser.

4.1.4 Parser Modifications

The parser modifications represent changes to the grammar rules (section 4.1.1.2). Each rule of the parser is composed of terminal and non-terminal symbols and user-defined procedures arranged in a very particular pattern. The procedures follow the C calling conventions and they allow the use of individual symbols as their arguments. Thus, if more functionality is desired, we have to either insert new functions into the rules,

or define new rules with their own functions. The placement of tokens and the function blocks is important since it determines the correctness of the parser.

The two most important parser modifications are those that track (1) the function definitions and (2) the construct declarations. The first modification is required so that we know when each function starts and when it ends. At the beginning of the function definition we initialize all the variables (like `ts_ptr`, `fts_ptr` and `cur_node`). At the end of function definition we need to collect all the timing information, print it in the intermediate file and deallocate all SLOTT structures.

The second modification is needed to record each construct of the source code, *i.e.*, the tool must know where each construct begins and where it ends so that it knows what instructions belong to the defined constructs. The modification actually consists of two function call insertions for each construct. The first insertion is done at the beginning of the construct definition, the second at the end — just like the first modification to the function definition.

The third modification consists of inserting a function into the FOR loop rule. The function is added into the code of user functions anywhere *after* the definition of all three FOR loop sub-expressions since the function has to know what the sub-expressions look like. More precisely, SLOTT has to know what the counter variable is, what the initial, update and final values are, what the final condition is and what the update operation is.

Finally, if the lexical analyzer recognizes the new reserved word, then the parser should include a rule that will recognize the token and properly start and end a timing block. This rule resembles any other rule for a similar construct like WHILE, DO, SWITCH.

4.1.5 Code Generator Modifications

The last modification has to be carried out on the code generator — the final block in the compilation process. This unit reads the information either from the parser or from the optimizer and produces the binary or assembly output. As mentioned before in section 4.1.1.1, some local optimizations are also carried out during this process.

These modifications are compiler specific. However, there are some basic points that have to be taken care of. First, the utility has to intercept each instruction as it is generated and written out. The utility has to obtain information not only about instructions but also about their operands. Some compilers provide a numeric code for both the instructions and the addressing modes of their operands; others provide the final or partial string of the instruction along with the encoded information for the operands. In any case, the part that decodes and records the timing of instructions has to be specifically designed for the compiler. If

the numeric code for the instructions is not available, then a scanner and a string comparator has to be built that will decode the instruction. In that case, the best approach is to design a hashing table that would produce the match on the first try. To reduce the number of entries in one table, it is advisable to separate co-processor instructions from the main CPU instructions, *i.e.*, produce a completely new table for FPU instructions if it is possible. This approach does not only introduce modularity into the design, but also it is more flexible and produces smaller hashing tables especially if the CPU and FPU instructions can be easily differentiated[2]. Since the code generator is compiler specific, the modifications will have to be tailored for it. The best and least intruding approach is to insert the fewest number of SLOTT instructions into the code generator and mimic the output function of the code generator to obtain the assembly instructions and their operands.

Another modification involves recovering the information about the program structure, *i.e.*, the program's constructs; otherwise, the machine analyzer will not know where the generated instruction is to be recorded. If the code is generated directly from the parsed nodes, the modification is very minimal; on the other hand, if any intermediate code is produced, the code generator has to recognize those special markers that have been saved during the parsing phase.

The following modifications are not necessary for SLOTT execution; however, they improve its usefulness and functionality. The code generator can be modified to: (1) record function names stored in registers, and (2) determine the value of constants used for immediate addressing modes. Since all of these additions are compiler specific, every unique compiler requires distinct modifications.

4.2 Machine Analyzer

The basic function of this unit is to record the timing information of each instruction that is received from the modified part of the code generator. Even though the idea is very simple, the actual implementation is hardware specific and may vary in its complexity. Consequently, we will describe the basic approach for determining the timing values rather than present one particular implementation.

4.2.1 Instruction Execution Timing Calculations

Machine analyzer consists mainly of tables and appropriate procedures that access these tables. The tables are hardware (*i.e.*, CPU/FPU) specific. Thus, the procedures are particularly designed to retrieve the

[2] All FPU instructions in 68000 family start with an 'f' character.

information from these tables. Within this context we will refer to low-level machine instructions as instructions.

The format that is used to store the timing information may vary quite substantially between different CPU and FPU types, even for units within the same family. For example, the non-cache CPUs, like 68000 and 68010, define only one timing value for each instruction [Motorola84]; while the cached types, 68020 and 68030, define more than one value. This is because the execution time of both CPUs is affected by the instruction cache, prefetching and overlap. Additionally, the timing value is influenced by the operand misalignment and the bus controller/sequencer concurrency of the instruction pipe [Motorola020-89], [Motorola030-89]. The 68030 timing values are also affected by the presence of the data cache and the on-chip MMU.

The simplest and basic formula used for calculation execution time of any instruction that is not affected by the cache and instruction overlap is:

$$T_{exec} = T_{insn} + T_{EA} \qquad [7]$$

where: T_{exec} is the execution time of the CPU instruction,
T_{insn} is execution time attributed to performing the instruction only,
T_{EA} is the time required to calculate the *E*ffective *A*ddress of the operand(s).

Formula [7] is directly applicable to the 68000 and 68010 CPUs. It is also used for the 68020 CPU even though the manufacturer defines three execution values – *best (BC), cache (CC)* and *worst case (WC)* – for both the instruction and effective address. The execution time for the timed code is then the sum of proper combination of these three cases. Even the [Motorola020-89] user manual recognizes this limitation:

> "... it is difficult to predict which timing table entry is used, since the influence of instruction overlap may or may not improve the BC/WC, or CC timings... it is also difficult to determine which combination of BC/CC/WC timing is required. Just how the instruction stream will fit and run with cache enabled, how instructions are positioned in memory, and the degree of instruction overlap are factors that are impossible to be accounted for in all combinations of the timing tables."

Since it is very difficult to accurately predict the correct combination that would be observed when executing an instruction stream, the tables can only be used to predict the best-case and worst-case time bounds[3] [Motorola020-89].

---

[3]This is not absolutely true, though, as we will see in Chapter 5.

The 68030 CPU takes into account the instruction prefetching and overlap more closely. The CPU manual elaborates how the timing information is gathered during the cached execution of an instruction stream. This calculation process considers the time information not only of the current instruction, but also, of the previous instruction. Consequently, two additional values are defined for each instruction: the *head* and *tail* time. This is reflected in the formula used for CC instruction execution time calculation [Motorola030-89]:

$$CC_1 + [CC_2 - \min(H_2, T_1)] + [CC_3 - \min(H_3, T_2)] + \ldots \qquad [8]$$

where: CC_n is the instruction cache case time for an instruction *n*,
H_n is the head time for an instruction *n*,
T_n is the tail time for an instruction *n*,
min (a,b) is the minimum of parameters a and b.

To complicate the calculation even further, the CC instruction time for most instructions is composed of the CC instruction time for the effective address calculation (CCea) overlapped with the CC instruction time for the operation (CCop). Then the more specific formula is [Motorola030-89]:

$$CCea_1 + [CCop_1 - \min(Hop_1, Tea_1)] + [CCea_2 - \min(Hea_2, Top_1)] + [CCop_2 - \min(Hop_2, Tea_2)] + [CCea_3 - \min(Hea_3, Top_2)] + \ldots \qquad [9]$$

where: $CCea_n$ is the effective address time for the instruction-cache,
$CCop_n$ is the instruction-cache time for the operation portion of an instruction,
Hop_n/Top_n is the head/tail time for the operation portion of an instruction,
Hea_n/Tea_n is the head/tail time for the effective address of an instruction,
min (a,b) is the minimum value of parameters a and b.

In our calculations we will use formula [7] that applies also to the 68020 CPU since formulas [8] and [9] apply to the 68030 CPU. The [8] and [9] formulas are mentioned only to show the complexity of cache time prediction. However, it is better to use more complex formulas than [7] since they describe the execution time more accurately especially if factors like cache and instruction overlap are involved. As we will see in chapter 5, it would have been more useful if Motorola had defined the execution time of 68020 in a similar way as they had defined the values for the 68030 CPU. Consequently, we will express the predicted execution time for 68020 with three values — *best*, *cache* and *worst*.

Some instructions' execution time depends largely on the value in their operand(s). These instructions are MOVEM (move multiple registers into and out of memory for register saving and restoration), ASL/ASR

and LSL/LSR (arithmetic and logical shift left/right), DIV (division) and MUL (multiplication). Some data books give the formula that describes the execution time in terms of operand values. In some cases these values are known to SLOTT (*i.e.*, the operand is a constant), thus it is possible to predict the execution value accurately. In most cases this value may be hidden in a register or a memory location. Thus SLOTT has to assume a certain value. Knowing the size of the operand, we can calculate the maximum execution time given the worst case scenario. However, it is not the maximum execution value that we will consider, but rather the average value that is derived from the minimum and maximum execution value. This way the calculated value will more closely predict the real execution time since the error magnitude will be smaller for minimum and maximum cases, and the overestimation and underestimation errors will have greater chance of canceling each other.

4.2.2 Recording Process

Although we refer to the effective address (EA) calculation as T_{EA}, the CPU may define several tables, each of which is used for particular classes of instructions. For example, *fetch immediate address* time is used with immediate instructions like `ADDI`, `SUBI`, `ANDI` etc., and *jump effective address* time is used in conjunction with `JMP` and `JSR` instructions. In addition to these tables, the 68020 has three more EA tables: *fetch effective address*, *calculate effective address* and *calculate immediate effective address*.

Instructions that use one or two operands usually define more than one execution value. Each value is specific for certain addressing modes of its operand(s), *i.e.*, one value may be defined for *register-to-register* operands, another for *register-to-memory* operands, and another for *memory-to-register* operands. All these different values must be included in the timing tables.

The actual calculation process is done in several steps that must be executed in an orderly way:

(1) Find the appropriate *instruction time table* (*ITT*).

(2) Determine the effective address (*EA*) of the operand(s).

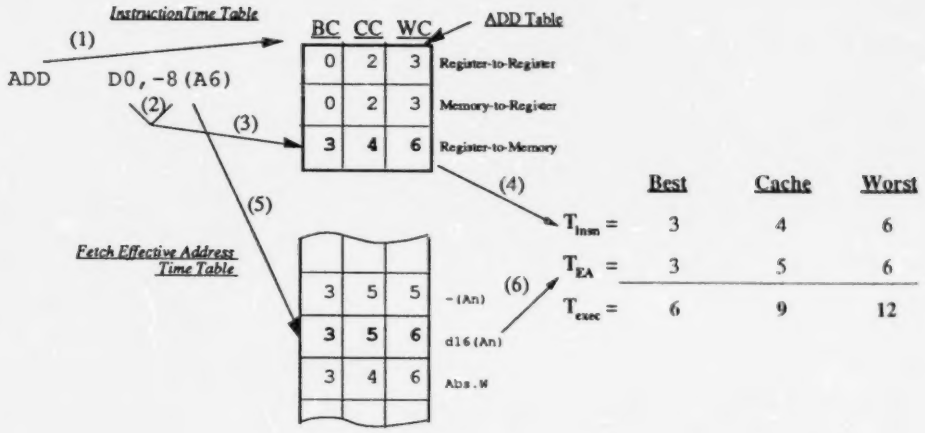
(3) If ITT has entries that refer to the EA(s) of instruction's operands, then determine the index into ITT from the operand EAs.

(4) Given the index, access the proper entry in ITT and assign it to T_{insn}.

(5) Retrieve the value from the effective address table for each of its operands and add them into the T_{EA} variable.

(6) Compute the predicted execution time using formula [7].

Figure 4.5 illustrates this process. We use a very simple assembly instruction that adds a value from the `D0` register to a local variable.



Steps:

(1) Determine the *Instruction Time Table* from the type of the instruction.
(2) Determine the addressing modes of all operands.
(3) Find out the proper index to the *Instruction Time Table* from the addressing modes.
(4) Look up the timing values from the *Instruction Time Table.*
(5) Depending on the type of addressing modes, if need be, consult appropriate *Effective Address Time Table* using the proper index.
(6) Look up the timing values from the *Effective Address Time Table.*

Figure 4.5 - Calculating the execution time of an instruction from the table values.

We can reduce the size of the timing tables by comparing the execution time of the instructions. The CPU instructions are grouped into several categories. Each category represents one class of instructions that perform similar operations and use the same number of operands. Moreover, some of these instructions in one class need the same number of CPU cycles. As a result, we can put such instructions into one group and use the same table entry for all those instructions in that group, *i.e.*, ADD, SUB, AND, OR and EOR instructions can use the same *arithmetic-logic instruction* (*ALI*) table to obtain the execution time. Thus in our previous example we would access the *ALI* table instead of the *ADD* table.

4.3 GCC Modification Aspects

The purpose of this section is to describe the required modifications to the GNU C compiler (GCC). We have modified the most current version available at the time (version 1.40) and targeted it for 68000 and 68020/68882 hardware platforms.

The most important reason we have chosen GCC is that it is the only compiler that has its source code available for modifications and distribution. Furthermore it is widely used within the scientific community and can be compiled for a wide variety of platforms and CPUs (Motorola, Intel, SPARC, RISC, MIPS, etc.).

4.3.1 Overview of GCC

GCC is a multi-pass compiler. Besides the basic components, it contains parts that allow multi-platform compilation and optimization — *intermediate code generator* (*ICG*) and *optimizer*. We will concentrate on the former component since the latter part is completely separated from SLOTT. The intermediate language is not only important for the multi-pass optimization, but also for the standardized generation of compilers for different hardware platforms. The advantage of this approach is that these compilers differ only in the code generator, the other parts stay the same.

The GCC's lexical analyzer and parser are implemented in a similar way as described in section 4.1.1. Consequently, their modifications follow the modification points mentioned in sections 4.1.3 and 4.1.4. The most significant part of the parser is its direct connection to the ICG instead of the code generator. Here we have to encode the program's structure into the intermediate language, make sure that it does not get altered or deleted by the optimization phase and decode it during the code generation phase.

The intermediate language is called a *register transfer language* or *RTL*. It describes the action of each instruction to be output in an algebraic form. Its structure is based on Lisp lists. It consists of an *internal form* (structures that point at other structures) and a *textual form* (strings used for machine description and debugging) that uses parentheses to express the internal structures in a hierarchical arrangement.

RTL supports four objects: *integers*, *strings*, *vectors* and *expressions*. The first two are self-explanatory, the third contains an arbitrary number of pointers to structures. The most important objects are the RTL expressions (RTXs) since they store the semantics of the source code. The most significant RTX is an `insn` expression that represents the code of a function in a doubly-linked list. However, GCC does not store all instructions in one type of `insn` RTXs. Rather, there are several types of `insn` expressions that store the information about the instructions, jumps, function calls and labels. The plain `insn` RTX is used for all instructions except jumps and function calls. The jumps are encoded in the `jump_insn` RTXs, and the function calls are contained in the `call_insn` expressions. Additionally, GCC defines one `insn` type that is used for debugging and declarative purposes — the `note_insn` or `note` instruction. The instructions of this type are particularly important in SLOTT's implementation since they do not affect the code generation; moreover, they are not removed or changed by the optimization phase.

The optimization phase consists of several passes; however, none of them influences SLOTT's operation or changes the code structure. Hence, SLOTT can faithfully represent the timing information of the generated code even if the optimization flag is turned on.

4.3.2 SLOTT's Program Structures and its Constructs in RTL

The RTL requires that the program structure and its constructs be encoded into the RTL list of instructions. The solution comes in the form of `note_insn` expressions. As has been outlined, these expressions are inserted by the compiler into the doubly-link list of instructions to indicate debugging and other information. During the optimization phase, the `notes` are not removed, neither are they shifted or re-allocated within the `insn` list. Furthermore, the optimization involves a reduction in the instruction count which is marked by changing unnecessary instructions from `insn` type to `note_insn` type.

Each `note_insn` expression carries information that describes its meaning or purpose. It contains two fields (besides pointers to neighboring instructions) of integer and string types. The integer field encodes the `note`'s description or meaning; the string carries more information whenever the type requires it. If the integer ID is positive, the number corresponds to the line number of the source code and the string contains the file name. If the ID is negative, the `note` has a special meaning. This includes the meaning of (1) a deleted instruction that has been changed from the `insn` to `note_insn` type and (2) the markers for scoping level boundaries or loops. Currently GCC does not use the string field if the ID is negative.

The `note` RTX is appropriately suited for the implementation of SLOTT markers. To accommodate the `note_insn` for our use, we have to define our meanings for it. This is done quite easily by expanding the negative ID range. Most of our markers require additional information to be stored with the markers like the time block's name, maximum iteration count or preferred jump. All this data is stored in the string format of `note_insn`'s string field.

4.3.3 Timing Process During GCC Compilation

As has been mentioned in section 4.3.1, the presence of RTL separates the timing process into two phases. However, this split confuses the order in which the timing information is gathered. The timing process becomes even more complicated when the `#pragma` markers are involved, especially if the `stiming` marker is used for timing a whole function.

Before we describe the actual process, it is necessary to mention that the compiler uses the linked list of RTL instructions only for the function *body*, while the prologue and epilogue parts are handled separately. Consequently, to record the timing value for the whole function, we have to initiate recording *before* the main function body is parsed! Assuming that the whole function is to be timed, the actual order of events is this:

(1) The lexical analyzer encounters the `stiming` marker before the function declaration. It immediately sets global variables to reflect that the timing block for the function has been defined.

(2) Since the function parsing has not commenced yet, the linked list of `insn` RTXs does not exist. Additionally, we would not gain anything if we inserted timing `notes` into the `insn` stream because only the function body exists in the link list format, not the prologue nor the epilogue.

(3) The whole function body is parsed, and the link list of RTL instructions is produced only for the current function.

(4) The function is optimized.

(5) Immediately after the optimization phase, a procedure is called that outputs the function's prologue. Since we have set up global variables in step (1) to indicate the start of function's `stiming`, the prologue will record the execution value of each instruction that it generates.

(6) The time information of each assembly instruction of the function body is recorded before the instruction is written out into the assembly file.

(7) Finally, a procedure for generating the function's epilogue is called which will record the timing information of its instructions.

4.4 Timing Analyzer

The analyzer is designed specifically for user needs. Consequently, we will present only the necessary details that pertain to the correct operation of the analyzer.

The timing analyzer is hardware specific. If the analyzer is used for different hardware platforms, then it has to acquire from the intermediate timing files the exact hardware configuration. This information has to include the CPU and FPU types along with their frequency.

Another requirement is to keep a list of all functions that have been analyzed so far. Currently, the analyzer does not suspend the calculation of the current block until an unknown function is found. Instead, it searches the list and if the function is there, it uses its timing information. If the function is missing, it just

assumes unbound value for that function even if it may appear later during the compilation process. This means that we have to declare functions in proper order, *i.e.*, follow Pascal style declarations. The present implementation is quite sufficient for our testing purposes. However, problems arise when a timing block contains functions defined in other source files or when the functions are not defined in the appropriate order. Then the analyzer could be updated to search for the functions in different files and suspend the analysis until the information of all functions is known.

Chapter 5

Evaluation of SLOTT

In order to know how well and accurate the timing tool predicts the execution time, a number of tests have been carried out. These test concentrate on certain aspects of the given hardware and program structures. For example, the CPU may have an instruction/data cache, and the floating point operations may be carried out by an FPU. First the "real" execution time of each important instruction was determined. Then the 68020 CPU was studied more closely. Finally, several smaller and larger procedures, or collection of procedures, were tested, and execution values compared against the SLOTT values. In order to demonstrate its flexibility, SLOTT has been modified and tested for other hardware configurations. Section 5.1 deals with the selection of appropriate hardware. Part 5.2 presents results of test programs run on a 68000 system. Section 5.3 discusses various factors that affect the execution time of a basic 68020 assembly instruction; section 5.4 presents the results obtained by system using the 68020/68882 CPU/FPU combination. Segment 5.5 gives the accuracy of SLOTT's predictions. Finally, part 5.6 compares the SLOTT performance against other tools discussed in section 2.2.3.

5.1 Choosing the Correct Hardware System

Even though the UNIX operating system provides means to collect timing information of any process, it is not suitable for performing accurate measurements. As a result, all tests have been performed on a dedicated card, Heurikon HK68/V2F, containing MC68020/68882 both operating at 16.67 MHz frequency. The card is connected to a Sun 3 workstation through a VME bus and can run any program compiled for the Sun 3 hardware. All Heurikon programs contain a small OS kernel that handles all the necessary OS calls including the timing calls and the selection of cache ON/OFF setting.

Programs were also run on a Macintosh II that has the 68020 CPU operating at 15.6672 MHz frequency. Macintosh lends itself as a good testing environment – its programs run in co-operative multitasking environment where each program decides when the context switch can occur and how long it should last. For the test programs, that we have used, no context switch was performed. Additionally, Macintosh programs can also disable all interrupts. Thus it can precisely time programs. The major stumbling block that prevents direct compilation of the UNIX assembly source files on Macintosh is the way in which the Macintosh environment deals with global variables. Macintosh programs access global variables through register A5, while UNIX uses absolute addressing. Execution values of several instructions were tested on both Heurikon and Macintosh. Since the Macintosh results closely followed the Heurikon results, a limited number of tests were performed on Macintosh to avoid the complexity of patching global variables.

The only accessible system that uses the 68000/010 CPU was a Macintosh SE operating at 7.8336 MHz. Again, we have excluded Sun 2 UNIX workstations for the reasons mentioned above. The only difference between the 68000 and 68010 is the virtual environment support and difference in timing values for some instructions. Having experience in running programs on the Macintosh II, we have modified the evaluation programs to exclude any global variables, and to use the local stack frame instead. This approach requires passing these local structures into the functions as pointer arguments. Since 68000 does not support an FPU co-processor, all programs that use floating point operations have been excluded.

5.2 Evaluation of MC68000 Hardware

The 68000 hardware system does not have any FPU support; consequently, all floating operations are performed in the Macintosh environment with the help of the SANE[1] library routines. To eliminate any library calls, all procedures involving floating point operations have been discarded. Also, procedures that perform long integer multiplication and division operations have been modified to use short integers since the MC68000 CPU does not support long integer format for these instructions.

The execution time of the 68000 follows formula [7] described in section 4.2.1. Additionally, the execution time is not affected by factors that affect more complex CPUs, like MC68020 (described later in section 5.3). The only factors that affect the MC68000 execution time are non-CPU related: speed of memory access, number of wait states, software timer interrupts, clock interrupts, memory refreshments, etc. These factors are quite substantial since they decrease the execution time of an instruction by 15% to 30%. Consequently, the execution time values of all instructions, involved in the test programs, have been

[1] Standard Apple Numeric Environment.

normalized to account for these factors. The normalization process involves calculation of actual execution time of an instruction directly from the execution time of test procedures.

5.2.1 Test Programs

All of these programs are identical to those mentioned in section 5.4 with some minor modifications that exclude the use of global arrays and variables (see section 5.1). Rather, these arrays are locally allocated in the `main()` procedure and passed as arguments into the test procedures.

We have used only four test programs: *Ins Worst*, *Bubl Worst*, *StrSrchW* and *Rand Num*. *Ins Worst* and *Bubl Worst* procedures perform insertion and bubble sort. We assume and test only the worst case, *i.e.*, the list of numbers arranged in descending order is sorted into ascending order. *StrSrch W* is a procedure that searches a test string of 512 bytes for a specific 6 byte string pattern of which none of the characters can be found in the test string. *Rand Num* has been taken from [Press90] and was changed to suit our needs. Actually, a loop inside the procedure has been expanded, while the loop count has been reduced. The procedure calculates random numbers using an array of integers. Since MC68000 CPU allows only short integer operations for multiplication and division instructions, all long integer variables have been re-defined into short ones to eliminate any library calls. In addition, GCC uses library routines for division and modulo calculations irrespective of operand sizes; therefore, it is necessary to define inline assembly macros to substitute these standard C operations.

5.2.2 Test Results

The tests are presented in Table 5.1. The predicted time values, generated by the timing tool, are given in the fields of *Predicted Time*. The average of actual execution time is given in *Average Time* field and the prediction error is indicated by *% Error* values. The sign of these error values indicates whether the predicted value underestimates (negative sign), or over-estimates (positive sign) the actual execution value.

	Ins Worst	Bubl Worst	StrSrchW	Rand Num
Predicted Time (s):	10.262	13.69	6.27	3.908
Average Time (s):	10.246	13.809	6.374	3.892
% Error:	0.15%	-0.86%	-1.63%	0.42%

Table 5.1 – Execution time of MC68000 test programs.

The predicted values are very close to the actual execution time. However, we have to mention one more characteristic that applies for some CPU instructions. The execution time of these instructions depends on the value present in one of its operands. For example, the execution time of the *shift*, *move multiple* and

multiplication instructions is defined to be proportional to the number of shifts, number of registers to be moved and number of bit changes in one of its operands. Our SLOTT can account for this difference only if the operand is immediate (a number). If the value is stored in a register or a memory location, SLOTT has to assume a certain value. In the *Rand Num* example, all source operands for multiplication have been compiled as immediate. But what value should the tool assume if the operand is not immediate — maximum, or an average?

To find out the best value, we have designed a small program, *TestMult*, that uses two arrays of short integers. These arrays are filled with random numbers and then each field of one array is multiplied with the field of the other array. The percentage error of using the average and maximum execution time is given in *% Error Ave Mult* and *% Error Max Mult* fields, respectively in Table 5.2.

Predicted Time of	TestMult
% Error Ave Mult:	-0.50%
% Error Max Mult:	6.18%

Table 5.2 – Testing the use of the average and maximum timing values for the MC68000 multiplication instruction.

The error values clearly show that the average value gives better prediction value by as much as 6%. These results confirm our first postulation of using average timing values for such instructions (section 4.2.1).

5.3 Factors That Affect Execution Time

The architecture of the MC68020 makes exact instruction timing calculations difficult due to the effects of: an on-chip instruction cache and instruction prefetch, operand misalignment, bus controller/sequence concurrency and instruction execution overlap [Motorola020-89].

The presence of instruction cache increases the speed of already fetched instructions in cache. Instruction prefetching allows faster execution since even if cache miss occurs, the bus controller prefetches the next instruction from the external memory while the microprocessor is busy with the current instruction execution. The CPU prefetches information in 4-byte, or long-word, blocks aligned on a long-word boundary. If two adjacent instructions happen to be only a word long, then the second instruction is already fetched and no additional read is necessary. If the prefetch falls on an odd-word boundary, due to a branch, the CPU will read the unnecessary *even* word along with the required *odd* word on one read cycle.

In these sections when we talk about a code size with a reference to cache performance, we actually refer to the code size of any loop construct. It is important to mention this peculiarity since only the code of a loop can profit from the positive effects of the cache, *i.e.*, no sequential instructions can execute faster unless they have been previously loaded into the cache.

Operand misalignment has impact on execution time when the CPU reads or writes into external memory since additional bus cycles must be performed to fetch the operand. The misalignment occurs when the address of a word operand falls across a long-word boundary, or a long-word operand falls on a byte or a word address that is not a long-word boundary.

Bus controller and sequencer can speed up the execution by running concurrently. The bus controller manages the bus activity. The sequencer, in turn, handles the bus controller and all internal processor operations such as calculation of effective address time and setting of condition codes.

Instruction execution overlap defines a time slot during which two instructions can execute concurrently. This time slot actually belongs to the previous instruction that can absorb the current instruction's execution time. [Motorola020-89] user's manual does not give these overlap values, but rather it gives the *best*, *cache* and *worst* case time values that account for the instruction overlap. On the other hand, the user's manual for MC68030, [Motorola030-89], defines the execution time in a more precise way by defining three variables for each instruction – the *head*, the *tail* and the total time that includes the overlaps. The effective execution time is calculated as the difference of the body and the smaller amount of either the previous instruction's tail or the current instruction's head (see section 4.2.1).

The following sections will show how some of these factors can affect the instructions execution time. Since it is difficult to test the bus controller/sequence concurrency, we concentrate on the other factors. All test programs, except the cache and operand alignment tests, have been run with only the cache OFF (COFF) setting. A test program (except of a cache test) consists of several procedures each having 100 move instructions with the same addressing modes.

5.3.1 Instruction Alignment

Even though it is not mentioned by Motorola in the list of the factors, the instructions alignment influences the instruction stream timing. [Motorola020-89] even gives some examples how the timing might be affected when 4-byte or longer instructions are not aligned on long word boundary. In the experiments, we have tested 4 and 8-byte instructions that have been properly aligned and misaligned. Results indicate that there are only some instructions that are adversely affected by this factor, while others are not affected at

all. The decrease in performance of the affected instructions ranges from 3.8% to 6.7% when the instructions are misaligned as compared to properly aligned instructions.

5.3.2 Instruction Overlap

This factor has been simulated by having 50 `move` instructions of one type interleaved with 50 `move` instructions of another type. The execution values of these "heterogeneous" blocks are compared to the predicted values derived from the execution values of the "homogeneous" blocks of the corresponding types. The results indicate that some instruction combinations do not differ from the "homogeneous" block values, while other combinations show improvement, and still others exhibit degradation. Both positive and negative results range from 1.5% to 7.3% in execution performance.

5.3.3 Effect of Operand Alignment

These tests have been performed with the cache ON (CON) and COFF settings. Only local variables and indexed variables have been used since GCC correctly aligns global variables. The results indicate that this factor affects both CON and COFF execution values in equal amounts, *i.e.*, the cache does not have any positive effect. The value of having a single operand misaligned accounts for additional 5 CPU cycles, both for CON and COFF cases. If two operands are misaligned, the overhead is 10 CPU cycles. This effect is quite substantial especially if the execution time of instructions with properly aligned operand(s) is small. The impact is even more significant under the CON setting. The worst degradation in performance reaches 80% when one operand is misaligned and 85% when both operands are affected (CON setting).

These results are quite instructional since they illustrate how little it takes to make a code run faster or slower, depending on a proper or haphazard way of declaring variables. The proper way of declaring variables is to make sure that all double and long-sized variables are declared first before any 2-byte and char-sized variables are declared.

The same applies for creating structures, *i.e.*, types with long-word boundary must be declared first, and then followed by other types. If a structure that contains long word fields is declared inside another structure, then we had better make sure that the first structure starts at a proper boundary. If there are more structures defined within another one, one should insert filler fields into any structures whose size is not a multiple of 4 bytes. If we do not adhere to these basic principles, then the predicted execution value of an instruction can be degraded by as much as 80% when only one operand is misaligned and by as much as 85% when both operands are affected! Consequently, this is quite a major factor that should not be ignored by any programmer, especially when programming *real-time* systems.

5.3.4 Effect of Cache

This section will explore the cache of the CPU, its effect on the execution time and correlation between code size and the execution time. The 68020 CPU on-chip instruction cache is a direct-mapped cache of 64 long-word entries [Motorola020-89]. It means that each block in memory can be placed only at one location in the cache, *i.e.*, the replacement algorithm determines the address in the cache from the memory address by a modulo arithmetic of the cache size. When the loop code size is smaller than the cache size, all instructions will stay in the cache, to result in a hit ratio of 1. If the code size goes beyond the cache size, the performance will decrease according to the following formula [Smith83]:

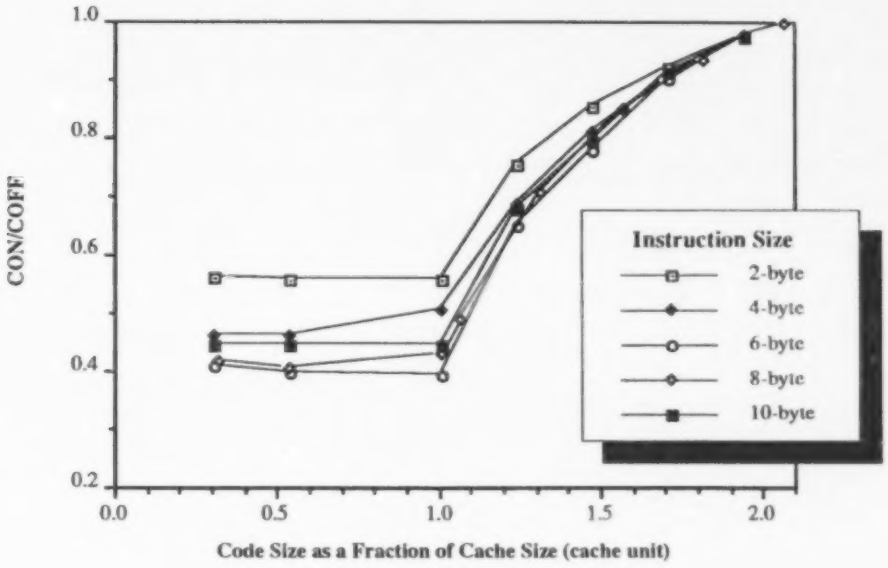
$$\text{if } M < N < 2M \text{ the block hit ratio is } \frac{2M - N}{N}; \qquad [10]$$

$$\text{if } 2M \leq N \text{ the block hit ratio is } 0,$$

where M stands for the cache size, and N is the loop size.

Thus, the cache miss occurs every time when the code size is larger than two cache size units.

All instruction sizes have been tested – from 2-byte to 10-byte. The test loop sizes range from 1/3 of the cache size to almost twice the cache size in several steps. The results have been obtained for both the CON and COFF settings. However, the final result is calculated as the CON/COFF ratio. This way we can demonstrate the effect of the cache on the instruction execution since the COFF value represents the absolute worst value of the CON setting. Graph 5.1 presents the execution results of all instruction sizes in a graphical form:



Graph 5.1 - The effect of loop size on the CON execution time.

The smaller changes in values of data below one cache unit size can be attributed to (1) the instruction alignment and (2) the influence of the instructions that make up the loop construct. Nevertheless, the results parallel formula [10].

5.3.5 Summary of the Results

The execution time is affected by several factors, some of which can be avoided by proper programming style, such as those that avoid operand misalignment; however, most of the factors are impossible to correct. Even the timing tool cannot track down and compensate for most of the aforementioned factors because they become visible only during the execution time. Consequently, we cannot predict the exact execution time of each instruction, especially if the instruction happens to reside in a loop (unless we go into great lengths to analyze the loop itself). Sequential instructions that are not a part of any loop will execute at COFF rates.

This concludes the testing of the factors that affect the CPU execution time. In the next section we will concentrate on general purpose and real-time programs and subroutines.

5.4 Testing General Purpose Procedures

Since our hardware contains an FPU, and the timing utility can deal with FPU instructions, we have divided the test programs into three groups – one that does not have any FPU instructions, another that spends about 50% of time executing FPU instructions, and the third group which uses mostly FPU instructions. It is important to mention that even a pure FPU code will require the CPU to calculate the effective address of the FPU operands. All tests have been run under both cache settings, and the predicted *best*, *cache* and *worst* case values have been compared with the average value of ten test runs.

5.4.1 Tests Involving only the CPU Instructions

Since cache is a major factor in the instruction execution, we have devised programs which employ small loops that fit inside the cache and programs that are significantly large to reduce the cache influence.

5.4.1.1 Significantly Large Programs

There are three such programs: *Rand Num*, *Encrypt* and *Vect*. *Rand Num* is identical to the *Rand Num* program presented in section 5.2.1. The only difference is that this time the multiplication and division operations are done on long integers.

Encrypt program "encrypts" random data using EOR (Exclusive OR) instructions. The data is encrypted in blocks of 8 bytes. Even though we could have used a FOR loop to EOR the data, we have expanded the instructions to increase the code size.

Vect procedure uses a couple of subroutines to calculate the dot product and cross product of two vectors. Each subroutine inside the *Vect* is called only once to eliminate the cache effects.

5.4.1.2 Small Programs

Five programs, *Ins Worst*, *Bubl Worst*, *StrSrch W*, *EncryptS* and *Rand NumS*, are small enough to fit inside the cache. *Ins Worst*, *Bubl Worst* and *StrSrch W* are identical to procedures mentioned in section 5.2.1.

EncryptS is the same procedures as the *Encrypt* procedure mentioned before, but in this case a FOR loop reduces the code of 16 EOR instructions into 2. *Rand NumS* is identical to *Rand Num* procedure; however, this one does not initialize the array of random numbers. Consequently, we need to call the *Rand Num* at the beginning of the program to initialize the array.

5.4.1.3 Cache OFF Test Results

The COFF results are presented in Table 5.3a. Each column to the right of the fist column contains data for one test program. The actual execution time is presented in the *Average* row. The percentage error of *best*, *cache* or *worst* prediction values, as compared to the average of actual execution values, are given in the appropriate *Error* fields. Since the cache is turned OFF, only the *worst* case values should be considered in the prediction. However, to indicate the spread of errors, all three cases are presented. This spread implies the accuracy of the prediction by the *worst* case values.

Cache OFF	Ins Worst	Bubl Worst	StrSrchW	Rand Num	Rand NumS	Encrypt	EncryptS	Vect
Average (s):	*11.07*	*15.455*	*16.31*	*9.69*	*12.465*	*19.865*	*19.925*	*20.325*
Error Best %:	-59.17%	-60.72%	-68.49%	-16.72%	-23.19%	-70.00%	-69.39%	-50.36%
Error Cache %:	-30.62%	-32.13%	-41.94%	0.00%	-4.61%	-37.48%	-37.31%	-32.55%
Error Worst %:	-11.47%	-11.94%	-20.42%	8.46%	7.74%	-19.25%	-16.54%	-14.10%

Table 5.3a – Execution time of CPU-only instructions (cache OFF case).

These results separate the programs into two groups according to the error spread between the *best* and *worst* case values — one that has division and multiplication instructions (*Rand Num* programs) and the other that does not (the rest). The first group shows the range of errors to be about 28%, while the range of the second group is about 50%. The smaller range of errors for the first group can be attributed to the small range of execution values between the *best* and *worst* cases for the division and multiplication instructions.

Since the execution time of division and multiplication operations depends on the value of one their operands, Motorola presents the execution times as the maximum number of CPU cycles required to perform these instructions. As a result, the predicted worst case values for both *Rand Num* programs (first group) show over-estimation by about 8%. On the other hand, the second group's *worst* case predicted values are underestimated by about 11.5% to 20%. This means that in order to predict the COFF case of the first group we have to *underestimate* the *worst* case value by about 8%, and of the second group we have to *over-estimate* the predicted *worst* case value by 15%.

The estimation errors are reduced when execution time of each instruction is normalized in the same way as in section 5.2. By normalizing execution time of every instruction we reduce the predicted trio of *best*, *cache* and *worst* values into one value that more closely reflects the real execution time. Even if some instructions are not normalized, the predicted range of *best*, *cache* and *worst* values will be smaller leading to more accurate prediction.

Three procedures have been tested: *Ins Worst, Bubl Worst* and *StrSrch W*. The results are given in Table 5.3b. The *Average* field gives the average execution time and the *Error* field indicates the prediction error using the normalized values.

Cache OFF	Ins Worst	Bubl Worst	StrSrchW
Average (s):	*11.07*	*15.455*	*16.31*
Error %:	-3.25%	-0.61%	-6.44%

Table 5.3b – Execution time of CPU-only instructions using normalized execution values. (cache OFF case).

The predicted values become very close to the actual execution values. Consequently, it would be advisable to normalize as many instructions as possible for the COFF case.

5.4.1.4 Cache ON Test Results

The CON results are presented in Table 5.3c. All fields correspond to the same fields of the COFF case. In fact, the predicted trio values are the same as in the COFF case, but in this case the *cache* results have to be used for comparison. The table fields correspond to the fields of Table 5.3a.

Cache ON	Ins Worst	Bubl Worst	StrSrchW	Rand Num	Rand NumS	Encrypt	EncryptS	Vect
Average (s):	*7.46*	*10.23*	*9.82*	*9.665*	*10.455*	*16.765*	*12.699*	*18.94*
Error Best %:	-39.41%	-40.66%	-47.66%	-16.50%	-8.43%	-64.45%	-51.96%	-46.73%
Error Cache %:	2.95%	2.54%	-3.56%	0.26%	13.73%	-25.92%	-1.65%	-27.61%
Error Worst %:	31.37%	33.04%	32.18%	8.74%	28.46%	-4.32%	30.96%	-7.81%

Table 5.3c – Execution time of CPU-only instructions (cache ON case).

From these results we can see how the error depends on the code size and the instruction type. The *Rand Num* and *Rand NumS* values again show small variation in error due to the abundance of multiplication and division/modulo instructions. However, *Rand Num* procedure is sufficiently long enough to decrease the effect of cache to the point that CON and COFF values are almost identical. On the other hand, *Rand NumS* procedure can fit inside the cache, thus its execution time is faster under CON setting than under COFF setting. The over-estimation of *cache* results can be attributed to the presence of multiplication and division instructions.

Other procedures with a small code, that fits inside the cache, (*Ins Worst, Bubl Worst, StrSrchW, EncryptS*) show very interesting results. Even though the range between the *best* error and the *worst* error is larger than in the COFF case, the *cache* results are very accurate to within ±3.5%! However, as the code size increases and the cache misses occur, the execution time becomes longer such that the *worst* case values should be considered instead (procedures *Encrypt* and *Vect*).

5.4.2 Tests Involving CPU and FPU Instructions in Same Proportions

5.4.2.1 Test Programs

These proportions are not expressed in the number of instructions, but rather in the number of CPU and FPU cycles. Since the FPU instructions on average take longer to execute than the CPU instructions, the program does not have to contain equal number of CPU and FPU instructions.

We have used only two test programs: *LinearSolve* and *FootMove*. *Linear Solve* has been taken from [Press90] and was adapted for our needs. The program uses three procedures, and it solves a linear set of equations. We have used random numbers as input data for each of the test runs. *FootMove* is a *real-time* program that calculates foot positions of a robot's leg as it moves from one position to the next. It uses Jacobian and its pseudoinverse to determine the incremental position of the robot's foot. The move was simulated between the same two positions. The iteration count in this case applies to the number of intermediate steps that comprise the move.

5.4.2.2 Test Results

Table 5.4 summarizes CON and COFF results. The COFF case section combines both data book and normalized values. Additionally, this table also shows the CPU cycle content (*% CPU*) using the *cache* timing values. The other fields correspond to the fields of Table 5.3a.

	Cache OFF				**Cache ON**	
	Data book Values		Normalized Values			
	FootMove	LinearSolve	FootMove	LinearSolve	FootMove	LinearSolve
% CPU:	45.40%	48.34%	50.57%	51.00%	45.40%	48.34%
Average (s):	*12.425*	*9.861*	*12.425*	*9.861*	*10.525*	*8.78*
Error Best %:	-33.52%	-24.55%	-4.63%	-6.60%	-21.52%	-15.26%
Error Cache %:	-10.82%	-8.53%	-1.49%	-3.56%	5.27%	2.73%
Error Worst %:	-0.85%	0.60%	0.04%	-2.04%	17.05%	12.98%

Table 5.4 – The execution time of programs having equal number of CPU and FPU instruction cycles. Both the data book and normalized values are given for the COFF case; the CON case uses only the data book values.

It is obvious from the table that the data book values are more spread. However, the magnitude is not as wide as for the CPU-only instructions. Also the *worst* case values are very close to the actual execution time. The normalized values are almost identical to the actual time of the *FootMove* program. The *Linear Solve* normalized value, on the other hand, is worse than the data book value. Consequently, it is hard to decide which timing values should be used. Since the data book values are closer to the actual values for

both cases, it might be better to use the values supplied by the manufacturer even though the *best-worst* spread is larger.

As the table indicates, the cache values should be used for prediction. However, this should not be surprising since the cache has been turned ON.

5.4.3 Tests Involving Mostly FPU Instructions

5.4.3.1 Test Programs

These procedures have been taken from a real-time program used for controlling a robot arm. They are *Puma Dir Kin* and *Puma Inv Kin*. Both procedures involve only floating point calculations and operations. The only CPU instructions present in the test procedures are used for moving data, accessing variables and calculating addressing modes of all operands.

Puma Dir Kin procedure determines direct kinematics of a Puma robot arm. It uses random numbers as input joint angles (Theta 1 through Theta 6) and calculates the arm's position and orientation in the 3-D space.

Puma Inv Kin procedure calculates the angle values for the robot's joints from the position and orientation values (indirect kinematics). It too uses random numbers to calculate the angles, but the randomness in this case is restricted to those values that place the end-effector into the working space of the robot. This procedure serves for two testing programs. The first program (*Puma Inv Kin*) utilizes the whole procedure, while the second program (*Puma Inv KinE*) simulates an error in the position/orientation values. This error causes the program to jump over the calculation of orientation angles (Theta 4 - 6), thus resulting in computation of only the position angles (Theta 1 -3).

5.4.3.2 Cache OFF Test Results

The COFF values are presented in Table 5.5a that compares the actual values against the data book timing values and the normalized values. The meaning of the table fields corresponds to the meaning of Table 5.4.

Cache OFF	Data book Timing Values			Normalized Timing Values		
	Puma Dir Kin	Puma Inv Kin	Puma Inv KinE	Puma Dir Kin	Puma Inv Kin	Puma Inv KinE
% CPU	8.11%	6.45%	6.81%	11.17%	8.42%	9.22%
Average (s):	*12.428*	*14.5255*	*10.953*	*12.428*	*14.5255*	*10.953*
Error Best %:	-6.34%	-1.90%	-1.12%	-1.27%	1.61%	3.26%
Error Cache %:	-1.83%	2.51%	3.26%	1.54%	4.71%	6.00%
Error Worst %:	1.71%	6.09%	7.00%	3.48%	6.78%	8.01%

Table 5.5a – Execution time of FPU-oriented programs (cache OFF case). Both normalized and data book prediction values are summarized.

Both data book and normalized values give very close results. This similarity can be attributed to the small percentage of CPU instructions. Since this is the COFF case, the *worst* case values should be considered. The over-estimation of the *worst* case values of *Puma Inv Kin* procedures indicates shorter execution path than the path considered during the analysis phase.

It is very difficult to say which case should be considered, but looking at the individual values, it would be more appropriate to take into account the *worst* case values. It seems that the *Puma Inv* procedures do not find all test conditions of IF statements TRUE since both of them exhibit shorter actual execution time as compared to *Puma Dir Kin* procedure which has almost the same fraction of CPU instruction cycles.

5.4.3.3 Cache ON Test Results

These results are present in Table 5.5b. The meaning of the table fields is identical to the meaning fields of Table 5.5a.

Cache ON	Puma Dir Kin	Puma Inv Kin	Puma Inv KinE
% CPU	8.11%	6.45%	6.81%
Average (s):	*12.4395*	*14.589*	*10.985*
Error Best %:	-6.43%	-2.32%	-1.41%
Error Cache %:	-1.93%	2.06%	2.96%
Error Worst %:	1.61%	5.63%	6.69%

Table 5.5b – Execution time of FPU-oriented programs. (cache ON case).

As in the COFF case, the error values are quite small. The spread is small due to the low percentage of the CPU instructions. *Puma Inv* procedures exhibit shorter execution time, thus the *worst* and *cache* case values overestimate the actual average execution time. In any case, the *cache* values, which should be used for the prediction, give the best results.

5.4.4. Summary of the Test Results

The results indicate that for a CPU with cache the execution time depends not only on the cache state, but also on the type of instructions and the code size itself. Whereas it is possible to predict quite accurately the sequential code that has no loops, it is impossible to predict exactly the execution time of a looping construct unless we analyze loops. This analysis might be hampered by subroutine calls from within the loop especially if the subroutine's address is hidden inside a register.

In general, we have observed from the results that the timing value approaches the *worst* case for the COFF setting, and the *cache* case for the CON setting, when the code gets larger and/or the FPU computation increases. This observation is useful since we do not have to be concerned with the loop size and can estimate the execution time directly from the predicted values.

The calculations also have to take into account instructions that tend to overestimate the execution time, like the `DIV` and `MUL` CPU instructions do. The execution time value of such instructions is defined by the manufacturer as the maximum.

To summarize the data, the range of errors for *best*, *cache* and *worst* cases has been displayed in Graphs 5.2a-c. The programs have been classified into the following six groups: *CPU/Small* (*Ins Worst*, *Bubl Worst*, *StrSrchW* and *EncryptS*), *CPU/SmallDM* (*Rand NumS*), *CPU/Large* (*Encrypt*, *Vect*), *CPU/LargeDM* (*Rand Num*), *CPU/FPU* (*LinearSolve* and *FootMove*) and *FPU* (*Puma Dir Kin* and *Puma Inv Kin*). The reason why the *CPU/Large* and *CPU/Small* have been further divided into those that contain division and multiplication operations and those that do not, is to demonstrate how these operations tend to decrease the error range while at the same time they tend to overestimate the execution time. Only three major groups have been tested with normalized COFF values – *CPU/Small*, *CPU/FPU* and *FPU*. The *Minimum error* bar indicates the minimum error value of all test programs in a group. The *Range of errors* bar shows how the errors range from the minimum to maximum value.

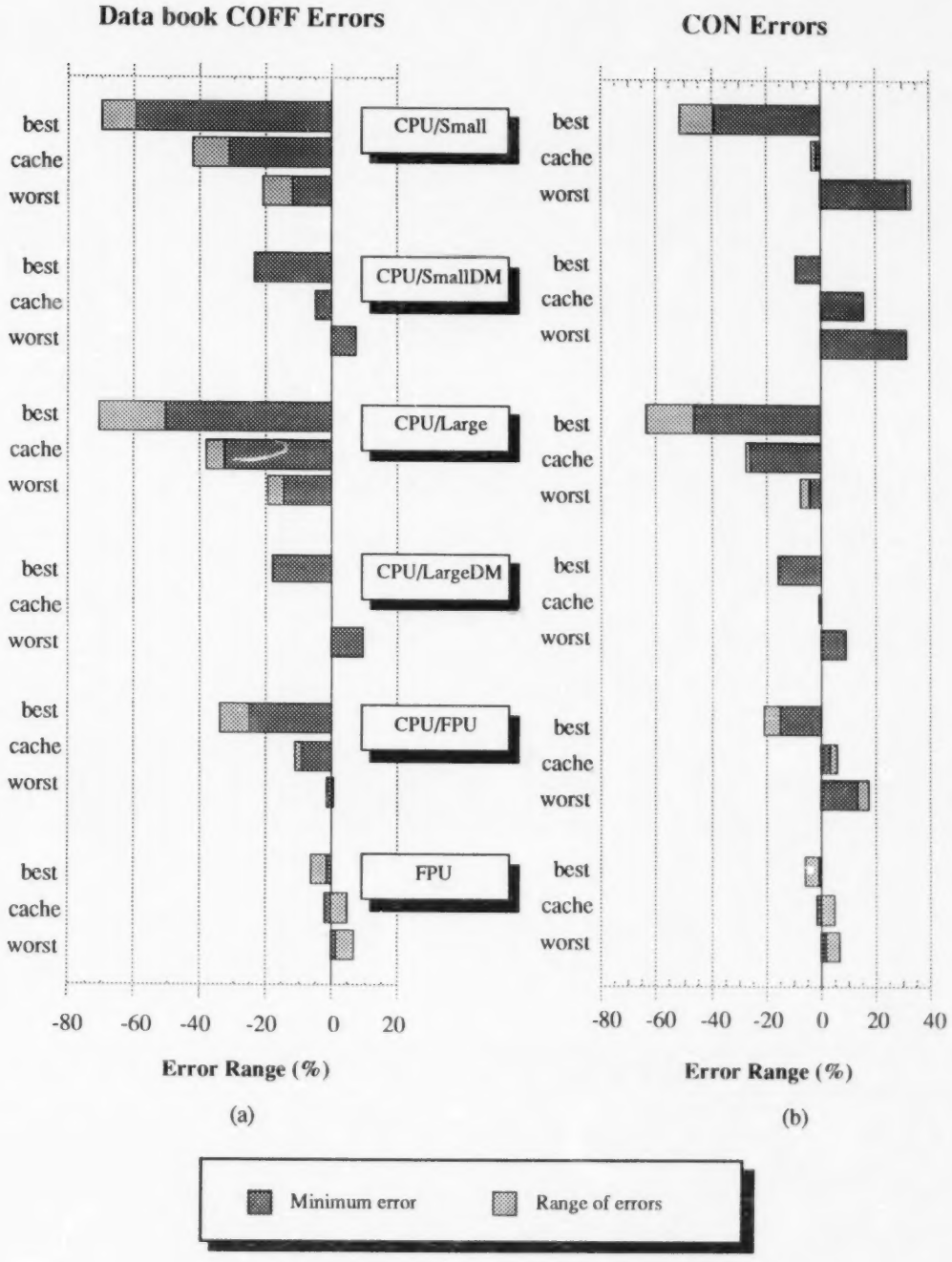
The *CPU/Small* group shows underestimation for COFF case of all data book values. However, the normalized values are very close to the actual execution time. The cache values for the CON case predict closely the execution time.

The *CPU/Large* group underestimates the execution time for both COFF and CON cases due to the code size that is longer than one cache size. Since the code is not longer than twice the cache size, some of the instructions remain in the cache resulting in smaller CON errors.

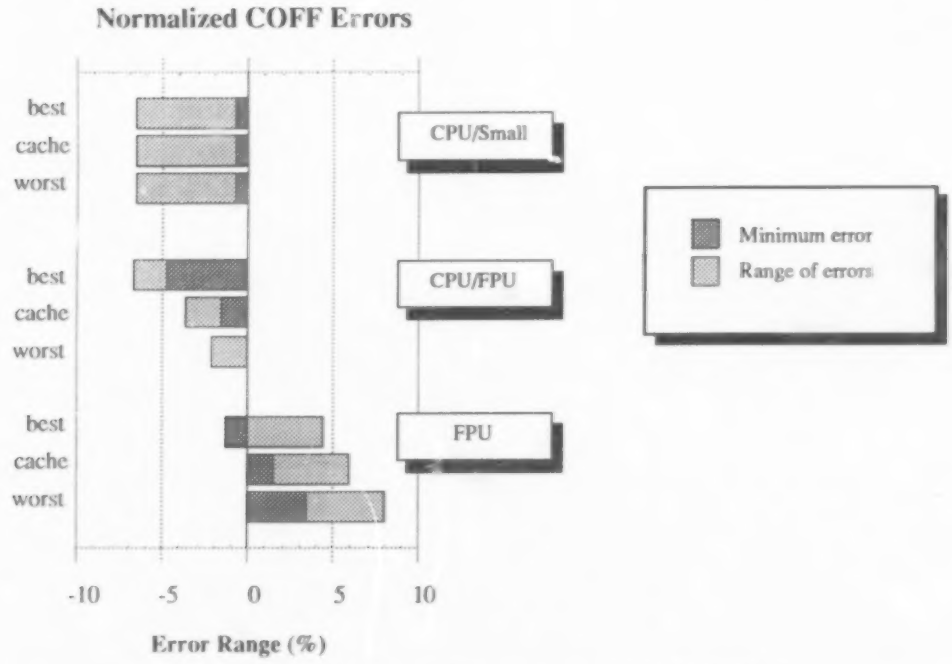
Both *CPU/LargeDM* and *CPU/SmallDM* groups indicate smaller range of CON and COFF errors. Additionally, they overestimate the execution values.

The *CPU/FPU* group shows smaller variation in error values than of CPU-only programs. The *worst* case data book and normalized values predict accurately the COFF execution values, while the *cache* case values are suited to predict the CON execution values.

The *FPU* group exhibits the smallest range of errors for both the COFF and CON cases. The high percentage of FPU cycles makes the execution time more predictable.



Graph 5.2a,b - Range of prediction errors using data book timing values.



Graph 5.2c - Range of prediction errors using normalized values.

5.5 Accuracy of SLOTT

The accuracy will be judged on the knowledge of the type of programs and the lack of this information. The type of programs is important only for MC68020 system since it is affected by various factors (section 5.3) that do not influence the MC68000 system.

5.5.1 MC68000 System

Once the instructions are normalized, the prediction is very precise. Using all four test program results, we can first calculate an average error offset. Using this offset value, the absolute error differences are calculated. These differences are then averaged to give the average error value for a predicted value that has been already offset. Table 5.6 summarizes the prediction errors and gives the average error and the offset error value. Both the error range and average error values are given in absolute terms. The offset error indicates whether the offset should be positive (to over-estimate) or negative (to underestimate the predicted value).

Error range (%)	0.15 to 1.63
Offset Error (%)	0.48
Average Error (%)	0.765

Table 5.6 – Range of errors, offset error and average prediction error of the MC68000 system.

These error values will apply to the programs of any type and any length. The execution time of instructions, that depends on the operand's value, is calculated either directly from the value, or as an average of the worst and best execution time values (section 5.2.2).

5.5.2 MC68020 System

These results are separated into COFF and CON error values. COFF case is further divided into results using data book values and normalized values. All offset error and average error values have been calculated using the same method as describe in previous section. Again we will categorize the programs into the six groups mentioned in section 5.4.4. To obtain the range of errors, at least two programs of the same type must be present. The number in the parentheses after the name of the group indicates the number of programs used for the calculation.

5.5.2.1 COFF Case with *Worst* Data Book Values

Since the cache is turned OFF, the code length is not a factor. Therefore, we can combine the *CPU/LargeDM* and *CPU/SmallDM* programs into one group to create *CPU/DM* category. Five groups have been compared and they are shown in Table 5.7a. The fields are the same as those for Table 5.6. Again, only the *Offset Error* value indicates with the sign whether the result should be over-estimated or underestimated.

	Error Range (%)	Offset Error (%)	Average Error (%)
CPU/Small (4)	11.5 to 20.4	15.1	3.4
CPU/Large (2)	14.1 to 19.25	16.7	2.6
CPU/DM (2)	7.74 to 8.46	-8.1	0.36
CPU/FPU (2)	0.6 to 0.85	0.125	0.73
FPU (3)	1.71 to 7.0	-4.9	2.15

Table 5.7a – Range of errors, offset errors and average prediction errors of the *worst* data book values under COFF setting.

As stated before, the *CPU/Large* and *CPU/Small* groups could have been combined into one group since the code length is not a factor. The results clearly show that both *CPU/Large* and *CPU/Small* categories

have almost the same error range, offset error and average error values. The division and multiplication operations tend to over-estimate the estimated result; therefore, the offset value of 8.1% is used to decrease the calculated value to arrive at better estimation. Both *CPU/DM* and *CPU/FPU* groups share the smallest average error, bellow 1%, once the results have been adjusted by the offset value.

5.5.2.2 CON Case with *Cache* Data Book Values

Here the code length is of utmost importance. Since *CPU/LargeDM* and *CPU/SmallDM* programs have only one program for each group, we cannot use them in calculation of the error values. The results are present in Table 5.7b and the fields correspond to the

	Error Range (%)	Offset Error (%)	Average Error (%)
CPU/Small (4)	1.65 to 3.56	-0.07	2.67
CPU/Large (2)	25.9 to 27.6	26.8	0.85
CPU/FPU (2)	2.73 to 5.27	-4.00	1.27
FPU (3)	1.93 to 2.96	-1.03	1.97

Table 5.7b – Range of errors, offset errors and average prediction errors of the *cache* data book values under CON setting.

The average error ranges between 0.85% to 2.67% once the results have been adjusted by the offset values. While cache values predict very closely execution time of *CPU/Small* programs (offset error is 0.07%), the prediction can be off by as much as 27% for *CPU/Large* programs. Consequently, it is desirable to compensate for this discrepancy by SLOTT during compilation.

5.5.2.3 COFF Case with Normalized Values

Only three groups are compared and evaluated: *CPU/Small*, *CPU/FPU* and *FPU* programs. The results are present in Table 5.7c and the meaning of the fields correspond to the meaning of Tables 5.7a,b.

	Error Range (%)	Offset Error (%)	Average Error (%)
CPU/Small (3)	0.61 to 6.44	3.43	2.00
CPU/FPU (2)	0.04 to 2.04	1.00	1.04
FPU (3)	3.48 to 8.01	-6.09	1.74

Table 5.7c – Range of errors, offset errors and average prediction errors of the *normalized* data book values under COFF setting.

The normalization process reduces the offset error and to some degree also the average error. The most significant effect of using normalized values can be observed for CPU-only programs. The effect on *CPU/FPU* and *FPU* programs is almost unnoticeable.

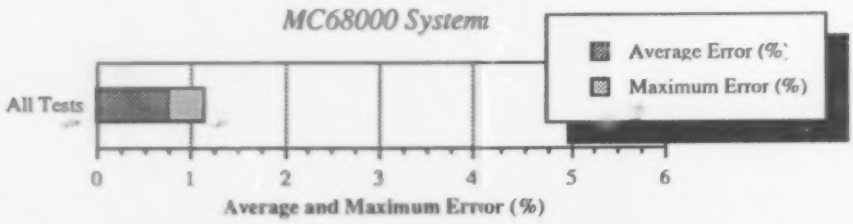
5.5.3 Summary

The average and the maximum errors of adjusted values are shown in Graph 5.3a and 5.3b. Graph 5.3a indicates prediction errors for MC68000 system. These errors apply to all programs of any type and length. Thus any predicted value will differ from the true execution time by 1.1% at maximum.

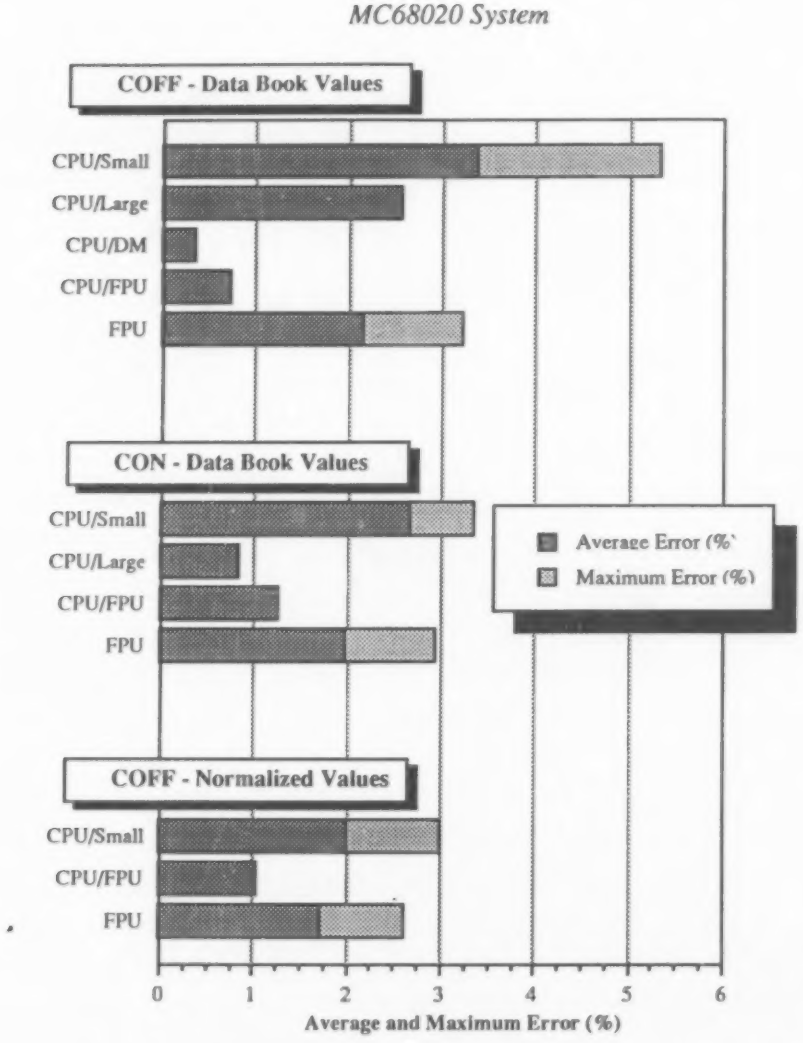
MC68020 system has to consider different factors, thus it is necessary to know what type of program is considered and whether cache is turned on. Using only data book values and knowing the type of programs, the maximum prediction errors range from 0.5% to 5.5% for the COFF case, and 0.9% to 3.5% for the CON case. If the normalized values are used, the maximum prediction error ranges from 1% to 3%.

If the program type is not known, then we have to have to assume the worst case that can occur. To arrive at the offset value, we have to refer to Graphs 5.2a and 5.2b. Under the COFF setting, we are using *worst* case prediction values. The largest error is exhibited by both *CPU/Small* and *CPU/Large* test procedures which is underestimation of 20%. Thus increasing the predicted *worst* case error by 20%, the over-estimation will shift from 10% (*CPU/DM* programs) to 30%. Under the CON setting, the *cache* values are used. Referring to Graph 5.2b, the largest error occurs for *CPU/Large* programs with the value of 25%. By shifting the base from 0 to the 25% mark, the over-estimation value for *CPU/SmallDM* program increases from 15% to 40%. Consequently, the *worst* result for the COFF case has to be over-estimated by 20%, while the *cache* value for the CON case has to be increased by 25%. However, the adjusted values can over-estimate the execution time by as much as 30% and 40% for the COFF and CON cases, respectively.

If the program type and the cache setting are not known,, then we have to assume the *worst* case values. Again, the offset is 20% (under COFF setting for CPU-only programs), but the over-estimation error will increase to 50% due to the *worst* CON error values of the *CPU/Small* and *CPU/SmallDM* programs (Graph 5.2b).



Graph 5.3a - Percentage errors for all MC68000 test cases.



Graph 5.3b - Percentage errors for MC68020 test cases.

5.6 Comparison of SLOTT with Other Systems

This section will try to compare the performance of SLOTT with the performance of tools described in section 2.2.3. The performance of source level timing tools depends on three major factors: (1) obtaining exact assembly instructions produced by the compiler, (2) the accuracy of aligning assembly instructions with language constructs and timing blocks, and (3) tool's features that can specify the correct program flow directly at the source level. MARS-C system provides markers that permit loop and code bounding at the source level, and RT Euclid accepts only bounded loops. Both RT Euclid and MARS-C obtain precise

assembly instructions. But the accuracy of instruction's alignment with language constructs can be questioned in the MARS-C system especially under optimization.

Even though RT Euclid CPU has a cache, Stoyenko does not describe how he deals with different loop sizes. Apparently, the author uses only value(s) present in the manufacture's data book. The test programs concentrate on how the utility can predict the execution time of the whole system. Thus several real-time programs have been tested under light, medium and heavy load conditions. To be fair to RT Euclid, we will assume only light load case values since they would correspond to SLOTT's predicted values. The claimed accuracy ranges between 2% and 5%. However, no indication is given of what the code size is and what the structure of these test programs is. The complexity of the test programs can be compared to our *FootMove* test program since we are timing a larger program consisting of several procedures. This characteristic is favorable for timing a system as a whole since over-estimation and underestimation errors of loops and non-looping code will tend to cancel each other out. Obviously, to scrutinize the prediction of RT Euclid more closely, programs consisting of individual procedures will have to be tested out. This evaluation is important to make sure that the tool can precisely analyze schedulability of individual tasks that are under time constraints.

The authors of MARS-C system do not give any comparison between the predicted and actual execution time. However, knowing that the tool has been designed for MC68000, we can say that the tool can be as accurate as SLOTT. The accuracy, though, depends on (1) the knowledge of jump sizes, (2) the treatment of instructions whose execution times are determined by the value of their operands, (3) the presence of selection markers and (4) alignment precision. Since the authors do not mention the first two points; the system does not support the third one, and might have problem with the fourth one, we can assume that MARS-C cannot surpass SLOTT in its performance.

Completely different approach in predicting the execution time is presented by Shaw and Park in source level timing schema (SLTS) [Park91]. As mentioned in section 2.2.3.1, SLTS does not obtain the exact generated code, rather it predicts the binary output. Consequently, it predicts large range of execution values depending on what size of atomic block is used for the prediction. For example, if one Large Atomic Block (LAB) is used for a function call, the predicted time is calibrated to the actual execution time. But if a number of Small Atomic Blocks (SABs) is used for the same function call, the predicted value can differ from the actual value by as much as 393% [Park91]. The authors have divided their test procedures from a simple expression that makes up one LAB, through simple procedures consisting of several lines of code, to complex procedures like an insertion sort and a scheduler. The *Insertion sort* procedure of SLTS parallels SLOTT's *Ins Worst* procedure. Considering the worst sorting case for both MC68000/010 systems, SLTS overestimates the execution time by 67% while SLOTT is accurate to within 0.15%! Although the authors categorize the *Scheduler* test procedure as a complex procedure, it consists of three procedures that are executed in a loop. Thus its complexity can be classified under the category of *Whole Real-time System.*

Table 5.8 gives an overview of the test results mentioned by the authors that developed their systems. As we can see, SLOTT outperforms SLTS and possibly MARS-C systems. Even though RT Euclid and SLOTT have been designed for different hardware systems, their accuracy in predicting the execution time of a real-time system is almost identical.

System Type	Simple Expression	Simple Procedure	Complex Procedure	Whole Real-time System
RT Euclid (0)	N/A	N/A	N/A	2% to 5%
MAXT/MARS-C	N/A †	N/A †	N/A †	N/A †
SLTS (1)	0% (LABS) •	0.72% to 9.7%	1.4% to 66.6%	31.2% to 81.5%
SLOTT	N/A ‡	0.15% to 1.6% (1)	0.15% to 1.6% (1)	2.7% to 5.3% (2)

Table 5.8 - Comparison of SLOTT with other source level timing tools.

Notes:

† No evaluation has been performed, but it may follow results of the SLOTT's MC68000 system.
‡ The error will depend on the normalization of execution time of instructions.
• The execution time of these simple expressions has been calibrated to the pre-timed pure execution time.
(0) NS16000/32000 CPU family results.
(1) MC68000/010 results.
(2) MC68020 results.

Chapter 6

Conclusion

The knowledge of precise time limits of real-time tasks, in time-constraint systems, is of utmost importance. By knowing the execution time of each procedure, the designer can more precisely fine-tune each portion of the source code. He can concentrate on the bottle-necks and parts of the code that are the most critical for the correct system performance. These execution time values can be obtained by two different means — testing or predicting.

The disadvantage of testing is that it requires a major portion of the development time. Programmers must allocate time to design test driver programs, and then analyze the data these programs generate. Furthermore, the designer might have to invest in expensive tools, like simulation programs, or hardware timing devices. Finally, the obtained execution timing may be affected by factors that are analogous to the Heisenberg uncertainty principle (section 2.2).

Reliable prediction within predetermined tolerances has definite advantages: (1) once implemented and calibrated it is very fast, (2) no test driver programs are required and (3) it allows the user to analyze different scenarios just by supplying different running conditions, *i.e.*, various loop iterations and distinct program flow paths. As a result, important timing information can be obtained before the code is actually run. The crucial factor that determines the quality of such a tool is how closely the tool can predict the actual execution time. This condition depends on the complexity of the hardware system and how closely the tool can resolve the factors that affect the execution time: cache, operand alignment, instructions overlap and alignment, etc.

This work presents a prediction timing tool – the *Source Level Oriented Timing Tool*, or *SLOTT* – that predicts the execution time directly form the source code. It is implemented as the part of a C compiler, but its simple and modular implementation allows easy import into other languages as well. The tool exists on its own and it does not restrain the compilation process. SLOTT gives the programmer the option to bound

loops, and segment of code and specify the program flow. It collects the timing information for the user-defined code fragments which can range from one line to the whole function. Since most real-time languages do not deal with the timing aspects of their programs, the functionality of such real-time systems can be enhanced by including SLOTT in their design.

SLOTT has been tested on two hardware platforms the 68000 and the 68020/68882. Results obtained for the 68000 CPU are very accurate, especially after the execution time of each instruction used in the test programs has been normalized. The normalization process is important because it takes into the account various hardware and system related factors like memory refreshing, additional wait states, timer interrupts and the DMA operations. The 68000 CPU does not have a cache and thus it is not affected by cache related factors as the 68020 CPU is.

The 68020 hardware system has a cache and is sensitive to operand/instruction alignment, instruction overlap and other factors. The cache OFF (COFF) case execution values are very close to the normalized COFF values for the CPU-only instructions. When the FPU instructions are included, the *worst* case values can be used to predict the COFF values. Most of the time; however, the CPU will be utilized with the cache ON (CON) setting. In this case it is very important to consider the loop size since the execution time of the CPU instructions can be significantly improved when the instructions are present in the cache. The FPU instructions are not affected as much since the T_{insn} portion of the T_{exec} time is more significant than the T_{EA} portion. Additionally, the FPU execution value of each instruction is defined by one data book value. This reduces the errors introduced by the triple timing values for the CPU. As the program size gets larger and the FPU content increases, the actual CON values get closer to the predicted *cache* values. Thus even though the loop size is important for small code segments, the execution time will average out for timing blocks that involve many functions of a variable code size.

Since it is very hard for a programmer to determine the loop size directly from the instruction stream, the next improvement of SLOTT should include this calculation. Knowing the loop size, SLOTT can then predict the execution time more precisely. This implies, that accurate COFF and CON values be determined for each instruction that is to be considered in the calculations of the variable loop size. The best approach to accomplish this task is to perform tests similar to tests described in section 5.3.4 where the CON/COFF ratio for slightly under-filled cache is calculated instead of a single CON value. Additionally, the COFF normalized values can be determined. The second test will establish all COFF values as a basis for the first test values that are used for the calculations of the CON values from the CON/COFF ratios.

Another requirement for the CISC (*Complex Instruction Set Computer*) CPUs is that the instruction size must be known. The 68000 family instructions range from 2 bytes to 10 bytes in size and it is fairly easy to determine the instruction size from the operand's addressing modes. Other architectures such as the

RISC (*Restricted Instruction Set Computer*) CPUs are even simpler in this regard since every instruction is 4-bytes long.

The current version of SLOTT assumes jumps to be of a certain length – average time of byte and word long jumps have been used. In most situations the difference between the actual size and the real one is insignificant, but in order to calculate the execution time more precisely, the jump size must be determined. This again requires that we keep track of the code size. Moreover, we have to create a link list of all instructions since some jumps may refer to forward labels in which case we have to find out the code size between the jump and the label before we can determine what the jump size is going to be. However, to know if the presumed jump size will correspond to the actual compiled size, we have to know whether the assembler will optimize the jumps, or whether it will blindly follow assembly instructions.

The markers can be diversified as well. For example, in some situations it would be easier to define the iteration count of the inner loops in terms of their relation to the outer loops. Then the bounding of the outer loop would automatically bound the inner loops as well. This way we could specify the iteration count of all loops involved in the matrix multiplication by associating the matrix size with the upper bound of the outer loop.

Similar improvement can be applied to the FOR loops. Here the inner loops could determine the relation of their iteration count to the iteration values of the outer loops. This means that SLOTT will have to recognize assignment and arithmetic/logic operations using, not just constants, but also the index variables of the outer loops. Furthermore, the variables of the outer loops would have to be accessible for the inner loops until the end of their definitions.

The selection markers could be expanded in their functionality. They could allow more than one alternative of the same conditional statement to be considered in the analysis phase. These alternatives could be specified statistically so that each of them would be bound to certain portion of total number of selections.

Finally, the tool can be expanded beyond the CISC architecture. These days the trend is towards the use of the RISC and the MIPS CPUs due to their design simplicity, increased performance and computational speed. It should not be then surprising that the current hardware real-time systems are migrating from the CISC to the RISC platforms. There are four major techniques employed in the RISC design that allow faster execution: (1) instruction pipelines, (2) load/store architecture, (3) delayed load instructions and (4) delayed branch instructions. Additionally, the architectural simplicity increases the performance due to the faster instruction decode, operation and access time [Kane88]. For example, many RISC CPUs define all instructions to be uniformly 32 bits long. This eliminates many of the misalignment problems mentioned in chapter 5 and simplifies the instruction decode algorithm. Most of the RISC processors claim to perform at

one CPU cycle per any instruction. Obviously, some of these features should prove advantageous in the RISC version of SLOTT designs; however, instruction pipeline, delayed load and branch instructions might cause problems for some SLOTT implementations. SLOTT design demands that the compiler keeps the construct markers at the precise positions since the delayed load and branch techniques can rearrange the assembly code in order to suit the RISC philosophy.

There is certainly a need for reliable, fast, and accurate source level timing tools like the SLOTT. With all the computing power at our hands the time has arrived for the tools that correctly calculate and predict the execution time directly from the source code. Using tools similar to SLOTT, resources spent on testing and simulation could be redirected to the development and improvement of program code, thus resulting in faster and more reliable performance. SLOTT provides a good base from which similar "SLOTT-like" production tools can evolve.

Bibliography

[Aho72] Aho A. V., Ullman J. D., *The Theory of Parsing, Translation, and Compiling*, Prentice-Hall, New Jersey, 1972.

[Aho77] Aho A. V., Ullman J. D., *Principles of Compiler Design*, Addison-Wesley, New York, 1977.

[Anderson88] Anderson R. L., "Investigating Fast, Intelligent Systems with a Ping-Pong Playing Robot", *Robotics Research, The Fourth International Symposium*, pp. 15-22, MIT Press, 1988.

[Barret79] Barret W. A., Cough J. D., *Compiler Construction Theory and Practice*, Science Research Associates, Chicago, 1979.

[Bauer76] Bauer F. L. and Eickel J., *Compiler Construction, An Advanced Course*, Springer-Verlag, Berlin, 1976.

[Ben-Ari82] Ben-Ari M., *Principles of Concurrent Programming*, Prentice-Hall, Englewood Cliffs, New Jersey, 1982.

[Bhandarkar91] Bhandarkar D., Clark D. W., "Performance from Architecture: Comparing a RISC and a CISC with Similar Hardware Organization", *Association for Computing Machinery*, ACM 0-89791-380-9/91/0003-0310, 1991.

[Biermann90] Biermann A. W., "A Simple Methodology for Studying program Time Complexity", *Computer Science Education 1*, pp. 281-292, 1990.

[Blackman75] Blackman M., *The Design of Real Time Applications*, John Wiley & Sons, New York, 1975.

[Brodie81] Brodie L., *Starting FORTH*, Prentice-Hall, New Jersey, 1981.

[Brodie84] Brodie L., *THINKING FORTH*, Prentice-Hall, New Jersey, 1984.

[Burns90] Burns A. and Wellings A., *Real-time Systems and Their Programming Language*, Addison-Wesley, New York, 1990.

[Buxton68] Buxton J. N., *Simulation Programming Languages*, North-Holland, Amsterdam, 1968.

[Campbell88] Campbell J., *C Programmer's Guide to Serial Communications*, Sams, Indianapolis, 1988.

[Carlisle85] Carlisle B., "Key Issues of Robotics Research", *Robotics Research, The Second International Symposium*, pp. 501-503, MIT Press, 1985.

[Cmelik91] Cmelik R. F., Shing I K., Ditzel D. R., Kelly E. J., "An Analysis of MIPS and SPARC Instruction Set Utilization on the SPEC Benchmarks", *Association for Computing Machinery*, ACM 0-89791-380-9/91/0003-0290, 1991, pp. 290-302.

[Cohen82] Cohen J., "Computer-Assisted Microanalysis of Programs", *Communications of the ACM*, Vol. 25, No. 10, Oct. 1982, pp. 724-733.

[Cooling91] Cooling J. E., *Software Design for Real-time Systems*, Chapman and Hall, London, 1991.

[Darlington78] Darlignton J., Burstall R.M., "A System which Automatically Improves Programs", *Programming Methodology*, Ed. by D. Gries, 1978, Springer-Verlag, New York.

[Dettmer88] Dettmer R., "Go Fast, Go Forth", *IEEE Review*, Vol. 34, No. 11, December 1988, pp. 423-426.

[Dimpsey91] Dimpsey R.T., Iyer R. K., "Performance Prediction and Tuning on a Multiprocessor", *Association for Computing Machinery*, ACM 0-89791-394-9/91/0005/0190, 1991, pp. 190-199.

[Donnelly88] Donnelly C, Stallman R. M., *Bison, The YACC–compatible Parser Generator*, October 1988.

[Fisher88] Fisher C. N., LeBlanc R. J., *Crafting a Compiler*, The Benjamin Cumings Publisher, 1988.

[Ganssle92] Ganssle J.G., *The Art of Programming Embedded Systems*, Academic Press, 1992, San Diego, California.

[Ghallab85] Ghallab M., "Task Execution Monitoring by Compiled Production Rules in an Advanced Multi-sensor Robot", *Robotics Research, The Second International Symposium*, pp. 393-401, MIT Press, 1985.

[Ghallab88] Ghallab M. and Alami R. and Chatila R., "Dealing with Time in Planning and Execution Monitoring", *Robotics Research, The Fourth International Symposium*, pp. 431-442, MIT Press, 1988.

[Gilb77] Gilb T., *Software Metrics*, Winthorp Publishers, Cambridge, Massachusetts, 1977.

[Graham82] Graham S. L., Kessler P. B., McKusick M. K., "gprof: a Call Graph Execution Profiler", *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, SIGPLAN Notices, Vol. 17, No. 6, pp. 120-126.

[Halang91] Halang W. A., Stoyenko D. S., "Constructing Predictable Real Time Systems", Kluwer Academic Publishers, Boston, 1991.

[Hanafusa85] Hanafusa H. and Inoue H., "Panel Discussion: Key Issues of Robotics Research", *Robotics Research, The Second International Symposium*, pp. 520-524, MIT Press, 1985.

[Harbison84] Harbison S. P., Steele G. L., *C a Reference Manual*, Prentice-Hall, New Jersey, 1984.

[Head64] Head R. V., *Real-time Business Systems*, Holt, Reinhart and Winston, 1964.

[Heath91] Heath W. S., *Real-time Software Techniques*, Van Nostrand Reinhold, New York, 1991.

[Hennessy90] Hennessy J. L., Patterson D. A., *Computer Architecture A Quantitative Approach*, Morgan Kauffman Publishers, San Mateo, California, 1990.

[Hilton63] Hilton A. M., *Logic, Computing Machines and Automation*, Meridian Books, The World Publishing Co., New York, 1963.

[Horspool86] Horspool N. R., *C Programming in the Berkeley UNIX Environment*, Prentice-Hall, New Jersey, 1986.

[Ishikawa90] Ishikawa Y., Tokuda H., Mercer C., "Object-oriented Real-time Design", Technical Report CMU-CS-90-111, Dept. of Computer Science, Carnegie Mellon University, March 1990.

[Jahanian86] Jahanian F., Mok A. K., "Safety Analysis of Timing properties in Real-Time Systems", *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 9, September 1986.

[Kane88] Kane G., *MIPS RISC Architecture*, Prentice-Hall, New Jersey, 1988.

[Katzan, 84] Katzan H., *Invitation To Ada*, Petrocelli Books, New York, 1984.

[Kelly-Bootle85] Kelly-Bootle S., Fowler B., *68000, 68010, 68020 Primer*, The Waite Group, Sams, Indianapolis, 1985.

[Kernighan88a] Kernighan B. W., Ritchie D. M., *The C Programming Language*, Prentice-Hall, New Jersey, 1988.

[Kernighan88b] Kernighan B. W., Ritchie D. M., "The State of C", *BYTE*, August 1988, McGraw-Hill, pp. 205-210.

[Klingerman89] Klingerman E. and Stoyenko A. D., "Real-time Euclid: A Language for Reliable Real-time Systems", *IEEE Transactions on Software Engineering*, pp. 941-949, Vol. SE-12, No. 9, September 1989.

[Leigh88] Leigh A. W., *Real-time Software for Small Systems*, Halsted Press, John Wiley & Sons, New York, 1988.

[Leui-1 82] Leui J., DeVlaminck K., Huens J., *A Programming Methodology in Compiler Construction, Part 1: Concepts*, North-Holland, New York, 1982.

[Leui-2 82] Leui J., DeVlaminck K., Huens J., *A Programming Methodology in Compiler Construction, Part 2: Implementation*, North-Holland, New York, 1982.

[Leventhal86] Leventhal L. A., *68000 Assembly Language Programming*, Osborne McGraw-Hill, New York, 1986.

[Lieglois85] Lieglois A. and Borrel P. and Dombre E., "Programming, Simulating and Evaluating Robot Actions", *Robotics Research, The Second International Symposium*, pp. 411-415, MIT Press, 1985.

[Lippman89] Lippman, S. B., *C++ Prime*, Addison-Wesley, New York, 1989.

[Liu91] Yu-cheng Liu, *The M68000 Microprocessor Family*, Prentice-Hall, New Jersey, 1991.

[Martin67] Martin J., *Design of Real-time Computer Systems*, Prentice-Hall, New Jersey, 1967.

[McKeeman70] McKeeman, W. M. Horning J. J., Wortman D. B., *A Compiler Generator*, Prentice-Hall, New Jersey, 1970.

[McLeod68] McLeod J., *Simulation*, McGraw-Hill, New York, 1968.

[MIPS89] MIPS Computer Systems, Inc., *UMIPS-V Reference Manual (pixie and pixstats)*, MIPS Computer Systems, Sunnyvale, CA, 1990.

[Motorola84] Motorola, *M68000 16/32-Bit Microprocessor Programmer's Reference Manual*, Prentice-Hall, New Jersey, 1984.

[Motorola-020 89] Motorola, *MC68020 32-Bit Microprocessor User's Manual*, Prentice-Hall, New Jersey, 1989.

[Motorola-030 89] Motorola, *MC68030 Enhanced 32-Bit Microprocessor User's Manual*, Prentice-Hall, New Jersey, 1989.

[MotorolaFPC89] Motorola, *MC68881/MC68882 Floating-Point Coprocessor User's Manual*, Prentice-Hall, New Jersey, 1989.

[Park91] Park C. Y. and Shaw A. C., "Experiments with a Program Timing Tool Based on Source-Level Timing Schema", *IEEE Computer*, pp. 48-57, May 1991.

[Polak81] Polak W., "Compiler Specification and Verification", *Lecture Notes in Computer Science*, Vol. 124, Springer-Verlag, Berlin, 1981.

[Pollack72] Pollack B. W., *Compiler Techniques*, Auerbach Publishers, New York, 1972.

[Pountain87] Pountain D., *Object-Oriented Forth*, Academic Press, Harcourt Brace Jovanovich Publishers, Orlando, Florida, 1987.

[Press90] Press W.H., Flannery B.P., Teukolsky S.A., Vetterling W.T., *Numerical Recipes in C*, Cambridge University Press, 1988-90.

[Ripps89] Ripps D. L., *An Implementation Guide to Real-time Programming*, Yourdon Press, Englewood Cliffs, New Jersey, 1989.

[Rogers84] Rogers M. W., *Ada: Language, Compilers and Bibliography*, Cambridge University Press, Cambridge, 1984.

[Sarkar89] Sarkar V., "Determining Average Program Execution Times and their Variance", *SIGPLAN Notices*, Vol. 24, No. 7, Jul. 1989, pp. 298-309.

[Shaw89] Shaw A. C., "Reasoning About Time in Higher-Level Language Software", *IEEE Transactions on Software Engineering*, pp. 875-889, Vol. 15, No. 7, July 1989.

[Schreiner85] Schreiner A.T., Friedman H.G. Jr., *Introduction to Compiler Construction with UNIX*, Prentice-Hall, 1985.

[Shumate88] Shumate K., *Understanding Concurrency in Ada*, McGraw-Hill, New York, 1988.

[Smith83] Smith J. E., Goodman J. R., "A Study of Instruction Cache Organization and Replacement Policies", *The 10th Annual International Symposium on Computer Architecture*, Vol. 11, No. 3, 1983.

[SPC89] The Software Productivity Consortium, *Ada Quality and Style, Guidelines for Professional Programmers*, Van Nostrand Reinhold, New York, 1989.

[Stallman88] Stallman R. M., *Internals of GNU CC* for version 1.21, April 1988.

[Stallman91] Stallman R. M., *Using and Porting GNU CC* for version 1.40, June 1991.

[Stephens91] Stephens C., Cogswell B., Heinlein J., Palmer G., "Instruction Level Profiling and Evaluation of the IBM RS/6000", *Association for Computing Machinery*, ACM 0-89791-394-9/91/0005/0180, 1991, pp. 180-189.

[Stoyenko87] Stoyenko A. D., "A Real-Time Language Analyzer With A Schedulability Analyzer", Technical Report CSRI-206, December 1987, Computer Systems Research Institute, University of Toronto, Canada.

[Stroustrup88] Stroustrup B., "A Better C?", *BYTE*, August 1988, McGraw-Hill, pp. 215-216D.

[Tzafestas84] Tzafestas S. G., *Digital Techniques in Simulation, Communication, and Control*, North-Holland, New York, 1984.

[USDoD83] United States Department of Defence, *Reference Manual for the ADA Programming Language*, Springer-Verlag, New York, 1983.

[Vajapeyam91] Vajapeyam, S., Sohi G. S., Hsu W. C., "An Empirical Study of the CRAY Y-MP Processor using the Perfect Club Benchmarks", *Association for Computing Machinery*, ACM 0-89791-394-9/91/0005/0170, 1991, pp. 170-179.

[Wirth86] Wirth N., "Microprocessor Architectures: A Comparison Based on Code Generation by Compiler", *Communications of the ACM*, Vol. 29, No. 10, Oct 1986, pp. 978-990.

[Young82] Young S. J., *Real Time Languages Design and Development*, Halsted Press, John Wiley & Sons, 1982.

[Young88] Young M., Taylor R. N., "Combining Static Concurrency Analysis with Symbolic Execution", *IEEE Transactions on Software Engineering*, Vol. 14, No. 10, October 1988.

[Yourdon72] Yourdon E., *Design of On-line Computer Systems*, Prentice-Hall, New Jersey, 1972.

Appendix

This appendix includes one test procedure from each test groups. The call from the `main()` procedure will be illustrated with the last example. The following is the *Puma Dir Kin* test procedure:

```
#include     "puma.h"

/*  Define inline assembly macros */

#define  sin(x)   \
     ({  float __value, __arg = (x);    \
              asm ("fsin%.x %1,%0": "=f" (__value): "f" (__arg));      \
          __value;     })

#define  cos(x)   \
     ({  float __value, __arg = (x);    \
              asm ("fcos%.x %1,%0": "=f" (__value): "f" (__arg));      \
          __value;     })

#define  atan(x)  \
     ({  float __value, __arg = (x);    \
              asm ("fatan%.x %1,%0": "=f" (__value): "f" (__arg));     \
          __value;     })

#define  sqrt(x)  \
     ({  float __value, __arg = (x);    \
              asm ("fsqrt%.x %1,%0": "=f" (__value): "f" (__arg));     \
          __value;     })


#define  asin(x)  \
     ({  float __value, __arg = (x);    \
              asm ("fasin%.x %1,%0": "=f" (__value): "f" (__arg));     \
          __value;     })

#define  acos(x)  \
     ({  float __value, __arg = (x);    \
              asm ("facos%.x %1,%0": "=f" (__value): "f" (__arg));     \
          __value;     })


#define  CD       180.0/M_PI
#define  CR       M_PI/180.0

Thetas   th, rth;
transf   tr;


#pragma  stiming ()

DirectKinematics(pos, ths)
Thetas   *ths;
transf   *pos;
[

     float    t1, t2, t3, tUniq, delta;
     float    c1, s1, c2, s2, c23, s23, c3, s3, c4, s4, c5, s5, c6, s6;
     float    U511, U512, U521, U522;
     float    U411, U412, U413, U421, U422, U423;
```

```
    float    U311, U312, U313, U314, U321, U322, U323, U324;
    float    U211, U212, U213, U214, U221, U222, U223, U224;
    float    *fpt, *fpp;
    float    ox, oy;
    float    ax, ay;
    float    px, py;

    fpt = &(ths->t1);
    fpp = &(pos->n.x);

    t1 = *fpt++;
    t2 = *fpt++;
    t3 = *fpt++;
    s1 = sin(t1);
    c1 = cos (t1);
    s2 = sin(t2);
    c2 = cos(t2);
    s23 = sin(t2 + t3);
    c23 = cos(t2 + t3);
    s4 = sin(*fpt);
    c4 = cos(*fpt++);
    s5 = sin(*fpt);
    c5 = cos(*fpt++);
    s6 = sin(*fpt);
    c6 = cos(*fpt);

    U512 = -c5*s6;
    U522 = -s5*s6;

    U412 = c4*U512 - s4*c6;
    U413 = -c4*s5;
    U422 = s4*U512 + c4*c6;
    U423 = -s4*s5;

    U222 = s23*U412 + c23*U522;
    U223 = s23*U413 + c23*c5;
    U224 = c23*d4 + s2*a2;
    U212 = c23*U412 - s23*U522;
    U213 = c23*U413 - s23*c5;
    U214 = -s23*d4 + a2*c2;

    ox = c1*U212 - s1*U422;
    oy = s1*U212 + c1*U422;
    ax = c1*U213 - s1*U423;
    ay = s1*U213 + c1*U423;
    *fpp++ = oy*U223 - ay*U222;   /* = nx */
    *fpp++ = U222*ax - ox*U223;   /* = ny */
    *fpp++ = ox*ay - oy*ax;       /* = nz */

    *fpp++ = ox;
    *fpp++ = oy;
    *fpp++ = U222;

    *fpp++ = ax;
    *fpp++ = ay;
    *fpp++ = U223;

    px = c1*U214 + s1*d2;
    py = s1*U214 - c1*d2;
    *fpp++ = px;
    *fpp++ = py;
    *fpp++ = U224;

/* Set global flags about the initial positions of th1, th2, th5     */
    myatan2 (py, px, tUniq);       /*  Macro for calculating atan2   */
    tUniq += M_PI_2;
    delta = tUniq - t1;
    if (delta < 0.0)
        if (delta > -M_PI)    /* Make sure abs(delta) < 180 degrees */
            th1L_R = LEFT;
        else
            th1L_R = RIGHT;  /* else it must be the other state     */
    else
        if (delta < M_PI)
            th1L_R = RIGHT;
        else
            th1L_R = LEFT;

    px = a2*c2 - s23*d4;
    myatan2 (U224, px, tUniq);

    delta = tUniq - t2;

    /* Suppose the arm is in RIGHT position    */
    if (delta < 0)    {
        if (delta > -M_PI)
            th2U_D = UP;
        else
            th2U_D = DOWN;
    }
    else     {
        if (delta < M_PI)
```

```
            th2U_D = DOWN;
        else
            th2U_D = UP;
    }

    if (th1L_R == LEFT)  {    /* Exchange if th1 is LEFT    */
        if (th2U_D == UP)
            th2U_D = DOWN;
        else
            th2U_D = UP;
    }
    if (s5 < 0)
        th5F_NF = FLIP;
    else
        th5F_NF = NOFLIP;

}
```

The next group of procedures represents the *LinearSolve* test program:

```
#define mMin(x,y) ((x < y) ? x : y)
#define mMax(x,y) ((x > y) ? x : y)
#define mMod(x,y) (x - ((x / y) * y))

#define  EQN_SIZE     3

#define  fabs(x)  \
    ({  float __value, __arg = (x);    \
        asm ("fabs%.x %1,%0": "=f" (__value): "f" (__arg));     \
      __value;    })


double   *vv;
double   *aa;
int      *index;

int cnt1, cnt2, cnt3, cnt4, cnt5, cnt6, cnt7, cnt8, cnt9;
int cnt10, cnt11, cnt12, cnt13, cnt14, cnt15;


#define noErr                0
#define kNullRowErr       1
#define kNullPivotErr     2


#pragma stiming ()
int LUDeComposition (double *a, int n, int *index)
{
    int i,j,k, imax, errLevel;
    double   big, sum, temp;

    errLevel = noErr;

    for(i=0;i<EQN_SIZE;i++) {
        big = 0.0;
        for(j=0;j<EQN_SIZE;j++)
            big = mMax(big,fabs ((*(a + (i*n) + j))));

        if(big==0.0) {
            errLevel = kNullRowErr;
            break;
        }
        *(vv+i) = 1.0/big;
    }


    if(errLevel == noErr) {
        for(j=0;j<EQN_SIZE;j++) {

#pragma max_ic (2)
            for(i=0;i<j;i++) {
                sum = *(a + (i*n) + j);

#pragma max_ic (1)
                for(k=0;k<i;k++) {
                    sum -= (*(a + (i*n) + k)) * (*(a + (k*n) + j));
                }
                *(a + (i*n) + j) = sum;
            }

            big = 0.0;

#pragma max_ic (2)
            for(i=j;i<EQN_SIZE;i++) {
                sum = *(a + (i*n) + j);

#pragma max_ic (2)
                for(k=0;k<j;k++)
                    sum -= (*(a + (i*n) + k)) * (*(a + (k*n) + j));
```

```
                *(a + (i*n) + j) = sum;
                if((temp=(*(vv+i))*fabs(sum)) >= big) {
                    big = temp; imax = i;
                }
            }

            if(j!=imax) {
                for(k=0;k<EQN_SIZE;k++) {
                    temp = *(a + (imax*n) + k);
                    *(a + (imax*n) + k) = *(a + (j*n) + k);
                    *(a + (j*n) + k) = temp;
                }
                *(vv+imax) = *(vv+j);
            }

            *(index+j) = imax;

            if((*(a + (j*n) +j))==0.0) {
                errLevel = kNullPivotErr;
                break;
            }


            if(j!=(n-1)) {
                temp = 1.0/(*(a + (j*n) + j));

#pragma max_ic (2)
                for(i=j+1;i<n;i++)
                    (*(a + (i*n) + j)) *= temp;
            }
        }
    }
    return errLevel;
}


#pragma stiming ()
void LUBackSubstitution (double *a, int n, int *index, double *b)
{
    int     i,ii,ip,j;
    double  sum;

    ii = -1;

    for(i=0;i<EQN_SIZE;i++) {
        ip = *(index+i);
        sum = *(b+ip);
        *(b+ip) = *(b+i);
        if(ii>=0) {


#pragma max_ic (3)
            for(j=ii;j<=(i-1);j++)
                sum -= (*(a + (i*n) +j)) * (*(b + j));
        } else
            if(sum!=0.0)
                ii =i;
        *(b+i) = sum;
    }

    for(i=(EQN_SIZE-1);i>=0;i--) {
        sum = *(b+i);
#pragma max_ic (2)
        for(j=i+1;j<EQN_SIZE;j++)
            sum -= (*(a + (i*n) +j)) * (*(b + j));
        *(b+i) = sum/(*(a+ (i*n) +i));
    }

    return;
}


#pragma stiming ()
int LinearSolve (double *a, double *b, double *x, int n)
{
    int i,j, errLevel;

    for(i=0;i<EQN_SIZE;i++) {
        *(x+i) = *(b+i);
        for(j=0;j<EQN_SIZE;j++)
            *(aa + (i*n) +j) = *(a + (i*n) +j);
    }


    errLevel = LUDeComposition(aa,n,index);
    if(errLevel == noErr)
        LUBackSubstitution(aa,n,index,x);

    return errLevel;
```

```
}
```

The following is the *Rand Num* procedure:

```
#define  M1   259200
#define  IA1  7141
#define  IC1  54773
#define  M2   134456
#define  IA2  8121
#define  IC2  28411
#define  M3   243000
#define  IA3  4561
#define  IC3  51349

long     ix1, ix2, ix3;
long     r[4];

#pragma stiming ()
int
ranl (idnum)
int idnum;
{
     long     temp;
     int j, iff = 0;

     if (idnum < 0 || iff == 0)     {
          iff = 1;
          ix1 = (IC1 - (idnum)) % M1;
          ix1 = (IA1*ix1 + IC1) % M1;
          ix2 = ix1 % M2;
          ix1 = (IA1*ix1 + IC1) % M1;
          ix3 = ix1 % M3;

          ix1 = (IA1*ix1 + IC1) % M1;
          ix2 = (IA2*ix2 + IC2) % M2;
          r[1] = (ix1 + ix2/M2)/M1;

          ix1 = (IA1*ix1 + IC1) % M1;
          ix2 = (IA2*ix2 + IC2) % M2;
          r[2] = (ix1 + ix2/M2)/M1;

          ix1 = (IA1*ix1 + IC1) % M1;
          ix2 = (IA2*ix2 + IC2) % M2;
          r[3] = (ix1 + ix2/M2)/M1;

     }

     ix1 = (IA1*ix1 + IC1) % M1;
     ix2 = (IA2*ix2 + IC2) % M2;
     ix3 = (IA3*ix3 + IC3) % M3;

   j = 1 + ((3*ix3)/M3);

     temp = r[j];

     r[j] = (ix1 + ix2/M2)/M1;
     return temp;
}
```

The last examples show couple of short CPU-only test procedures along with the function calls from the `main()` procedure:

```
int array[ARR_SIZE];


#pragma stiming()
InsertSort ()          /*       Ins Worst        */
{
     int i, j, v;

     for (i=1; i<ARR_SIZE; i++)     {
          v = array[i];
          j = i;

#pragma  total_ic (124750)
          while (array[j-1] > v)     {
               array[j] = array[j-1];
               j--;
          }
          array[j] = v;
     }
}


#pragma stiming()
```

```
BubbleSort ()          /*       Bubl Worst        */
{
    int j, t;

#pragma total_ic (500)
    do  {
        t = array[0];
        for (j=1; j<ARR_SIZE; j++)          {
#pragma total_ic (124750)
            if (array[j-1] > array[j])      {
                t = array[j-1];
                array[j-1] = array[j];
                array[j] = t;
            }
        }
    } while (t != array[0]);

}


#pragma stiming()
WorstList ()           /*  Create array sorted in descending order     */
{
    int i;

    for (i=0; i<ARR_SIZE; i++)
        array[i] = 1000-i;
}


main ()
{
    register int     i, j;
    int start, end, k, l, m;


    for (k=0; k<2; k++)   {
        if (k==0)     {
            CacheOFF();
            printf ("CacheOFF\n");
        }
        else     {
            CacheON();
            printf ("CacheON\n");
        }
        for (l=1; l<=10; l++)     {
            printf ("Loop%d\n", l);

            j= 10;
            start = gettime ();

#pragma stiming (WIns)          /*  Time out Ins Worst procedure       */
#pragma max_ic (10)
            for (i=0; i<j; i++)   {
                WorstList ();
                InsertSort ();
          }
#pragma etiming (WIns)
            end = gettime ();
            printf ("%d\t", (end-start));

            j= 5;
            start = gettime ();
#pragma stiming (WBlb)          /*  Time out Bubl Worst procedure      */
#pragma max_ic (5)
            for (i=0; i<j; i++)   {
                WorstList ();
                BubbleSort ();
            }
#pragma etiming (WBlb)
            end = gettime ();
            printf ("%d\t", (end-start));

        }
    }
}
```